THE UNIVERSITY
of EDINBURGH

ENHANCING PRODUCTIVITY AND PERFORMANCE PORTABILITY OF OPENCL APPLICATIONS ON HETEROGENEOUS SYSTEMS USING RUNTIME OPTIMIZATIONS

THIBAUT LUTZ

Doctor of Philosophy

School of Informatics

Institute of Computing Systems Architecture

University of Edinburgh

2015

Enhancing Productivity and Performance Portability of OpenCL Applications on Heterogeneous Systems Using Runtime Optimizations

ABSTRACT

Initially driven by a strong need for increased computational performance in science and engineering, heterogeneous systems have become ubiquitous and they are getting increasingly complex. The single processor era has been replaced with multi-core processors, which have quickly been surrounded by satellite devices aiming to increase the throughput of the entire system. These auxiliary devices, such as Graphics Processing Units, Field Programmable Gate Arrays or other specialized processors have very different architectures. This puts an enormous strain on programming models and software developers to take full advantage of the computing power at hand. Because of this diversity and the unachievable flexibility and portability necessary to optimize for each target individually, heterogeneous systems remain typically vastly under-utilized.

In this thesis, we explore two distinct ways to tackle this problem. Providing automated, non intrusive methods in the form of compiler tools and implementing efficient abstractions to automatically tune parameters for a restricted domain are two complementary approaches investigated to better utilize compute resources in heterogeneous systems.

First, we explore a fully automated compiler based approach, where a runtime system analyzes the computation flow of an OpenCL application and optimizes it across multiple compute kernels. This method can be deployed on any existing application transparently and replaces significant software engineering effort spent to tune application for a particular system. We show that this technique achieves speedups of up to 3x over unoptimized code and an average of 1.4x over manually optimized code for highly dynamic applications.

Second, a library based approach is designed to provide a high level abstraction for complex problems in a specific domain, stencil computation. Using domain specific techniques, the underlying framework optimizes the code aggressively. We show that even in a restricted domain, automatic tuning mechanisms and robust architectural abstraction are necessary to improve performance. Using the abstraction layer, we demonstrate strong scaling of various applications to multiple GPUs with a speedup of up to 1.9x on two GPUs and 3.6x on four.

LAY SUMMARY

In recent years, the diversity and complexity of processing units embedded in electronic devices have grown considerably. Any system now contains a central component composed of multiple processing units – so called multi-core processors –, and a multitude of complex satellite co-processors specialized for specific tasks – such as graphics processing; such systems are called heterogeneous.

Each type of device has a very different architecture and typically a very different interface for software developers to interact with. This puts an enormous strain on programmers and considerably increases the complexity of the application source code.

Recently, new programming models have been developed to tackle portability issues. They present a unified interface for programmers, who can develop applications regardless of the hardware executing it. However these new models are hard to reason about and cannot fully exploit the intricacies of complex hardware. This leads to two common misuse patterns of these programming models in practice: underutilization and overutilization.

Underutilization is characterized by the exploitation of only a small subset of features available in the programming models, either because programmers think their application or hardware will not benefit from more advanced features, or because it increases the complexity of the code, which in turn impairs the development process. The programmer is trading decreased performance for improved productivity.

Overutilization is the reverse: expert programmers are trading decreased productivity and portability for better performance. Knowing the hardware components of the targeted system enables specialization of many aspects of the implementation, some of which improve performance but degrade performance when transferred to other systems.

The work presented in this thesis reconciles productivity and performance by tackling the identified misuse patterns. Underutilization of heterogeneous programming models is addressed with a transparent, self optimizing framework, allowing programmers to focus on productivity without sacrificing performance. At the other end of the spectrum, misuses are replaced with portable techniques, which automatically bridge the gap between what the programming models can express and hardware specific behaviors, preventing programmers from painstakingly tuning their code for specific systems.

DECLARATION

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Thibaut Lutz, Christian Fensch, and Murray Cole "PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems". In: *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013
- Thibaut Lutz, Vinod Grover "LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms". In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC'14)*, 2014
- Thibaut Lutz, Christian Fensch, and Murray Cole "Helium: a Transparent Inter-kernel Optimizer for OpenCL". In: *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU 8)*, 2015

Thibaut Lutz, May 22, 2015

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	Application Programming Interface	6
APU	Accelerated Processing Unit	11
CPU	Central Processing Unit	1
DLP	Data Level Parallelism	14
FPGA	Field-Programmable Gate Array	11
GPGPU	General-Purpose computing on Graphics Processing Unit	2
GPU	Graphics Processing Unit	2
ILP	Instruction Level Parallelism	14
ISA	Instruction Set Architecture	10
LLVM	Low Level Virtual Machine	35
MIMD	Multiple Instruction, Multiple Data	13
SIMD	Single Instruction, Multiple Data	13
SIMT	Single Instruction, Multiple Threads	13
SPIR	Standard Portable Intermediate Representation	35
TLP	Thread Level Parallelism	14

1 INTRODUCTION

Nowadays, heterogeneous systems are all around us – from the largest supercomputers to low end mobile phones. This section introduces how these systems arose and what benefits but also challenges they bring. The contributions of this thesis to tackle these problems are then described and an overview of this document is provided.

1.1 UBIQUITY OF HETEROGENEOUS SYSTEMS

The desire and need to process a growing amount of data at ever-increasing speed put enormous pressure on industries to produce more performant hardware. However, the increasing operational frequency of Central Processing Units (CPUs) at each hardware generation, which had for decades been the main focus of chip manufacturers and a gift for programmers wanting more performance effortlessly, was not able to cope with expectations anymore. It was the end of Dennard scaling [Den+74].

Dennard's scaling prediction states that by reducing the size and electrical characteristics of transistors, a proportional gain in density and operating frequency is achievable. This effect is known as process scaling. It applied from the early days of microprocessors in the seventies to the beginning of the twenty-first century: every two or three years the amount of transistors in microprocessors doubled, as foreseen by Moore [Moo65] in the sixties, and in the meantime the frequency was increasing by 40%.

However, in the aughts, manufacturers started to hit physical limitations in scaling down the transistor size. Leakage, heat dissipation and quantum effects meant that higher frequencies were no longer safe for reliable chips.

Nonetheless, Moore's law was still true and the number of available transistors kept increasing. Rather than making the size of the chip larger to increase the frequency, manufacturers started to duplicate the design of the microprocessor on a single die: this was the beginning of the multi-core processor era.

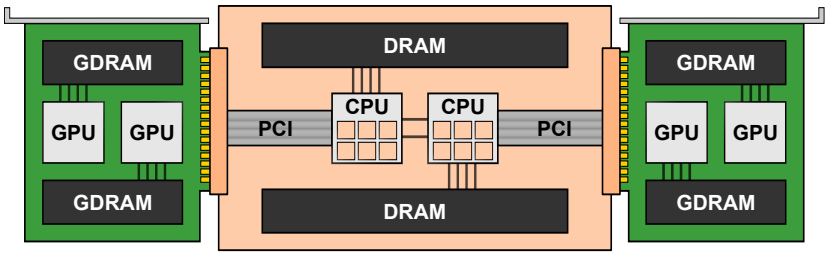


Figure 1.1: Example of system used in this thesis. The system is composed of a dual socket motherboard with two Intel Xeon E5-2620 processors and two dual GPU Nvidia Titan Z graphics cards. It supports a total of 24 parallel CPU threads and 11,520 parallel GPU threads.

This trend quickly gained increasing momentum. From a single core to two, two to four, and to eight. Some chips known as *many-core* processors now provide hundreds of cores on a single die [Pol; Phi].

In parallel, manufacturers developed more specialized hardware running alongside the main processor to lighten its workload. These single-purpose chips typically have a much simpler core design and better power efficiency compared to multicore processors; and often provide better performance for the tasks they are intended for. Digital Signal Processors were early example of such devices; they have been used since the eighties to improve efficiency of analog signal measurement and filtering.

More recently, Graphics Processing Units (GPUs) were developed to facilitate image processing and rendering. Driven by the needs of gaming and engineering industries, they picked up a very fast evolution since their introduction in the early eighties. The last decade has seen the emergence of a new type of computing technique known as General-Purpose computing on Graphics Processing Unit (GPGPU), where processing normally performed on CPUs is offloaded to a graphics processor.

Compute systems are now composed of a plethora of chips with varied architectures, each having widely different application domains and performance. Nowadays, any computer or embedded system, such as a mobile phone, contains at least one multi-core CPU, several GPUs and multiple single purposed accelerators, with a tremendous raw compute power. Figure 1.1 shows an example of a larger heterogeneous system composed of a total of two CPU chips and four GPUs. A vast amount of parallelism is available, with two dozen concurrent CPU threads and thousands of GPU threads. The combined compute power of these devices amounts to 16.5 Tera FLOPs, or the equivalent of over two thousands Intel Pentium 4 processors. However, this performance is very hard to tame in a single application.

1.2 IMPLICATIONS FOR PROGRAMMABILITY

The accelerating trend towards increasingly heterogeneous systems outpaces our ability to develop efficient software development methods. The gap between theoretical performance and actual utilization is growing rapidly due to the strain put on programmability.

THE THREE P'S Software programmability can be gauged as a combination of three factors: performance, portability and productivity.

Achieving good *performance* implies efficient utilization of the resources at hand – in terms of compute power or energy. For recent architectures, this goal can only be achieved through parallelism, for which scalability is also an important performance measure.

Portability is a key issue for heterogeneous systems. Execution portability allows the same program to adapt to different systems, either via programming languages or binaries, while performance portability should guarantee an efficient utilization of any device.

Of the three factors, *productivity* is the most affected by the emergence of heterogeneous systems – partly because it drives the other two. Software design should produce simple, concise, reusable and maintainable code. Implementing such code should require only a reasonable amount of expertise and hardware should not be a consideration.

Sadly, increasing the system heterogeneity stretches these principles to the point where they cannot all be considered. Worse, they all seem to contradict each other. Performance is inherently tied to hardware features, which immediately rules out performance portability. Achieving either portability or performance requires tremendous engineering efforts and highly specialized skills, which are considerable impediments to productivity. Hence increasingly heterogeneous systems are the cause of an impending software crisis.

SALVATION BY ABSTRACTION New programming models have been developed to address some aspects of these programmability issues. In particular, similarities across software or hardware patterns lead to the implementation of common software methods and principles aiming to reconcile the three Ps. These guidelines can be turned into languages and frameworks, which aim to tame heterogeneity by design. OpenCL is an example of such advances. As the first widely adopted solution for heterogeneous development, it is a giant leap towards improved programmability. Yet, in spite of providing reliable execution portability, these models are not very easy to use and still have limitations.

Disparities between the optimization strategies across devices prevent universal performance portability in the models mentioned above. Programmers still need to resort to substantial engineering efforts to use heterogeneous systems efficiently. This last point is aggravated by the typically low level abstraction, which re-introduces a dichotomy between productivity and performance, leading to different uses of the portable models.

USES OF PORTABLE MODELS Prioritizing productivity and using only a subset of the available features is largely observed in practice. Programmers do not need to understand the intricacies of the model; and they can manage to harness some power from the devices on hand. While this approach offers good productivity, it is largely suboptimal in terms of performance. This pattern characterizes a model *under-utilization*.

Efficient utilization of the model requires more effort to understand it. Reasonable productivity is achievable but limited by the set of skills required. Well exploited programming models allow graceful scaling to all available resources, which improves performance. However, this *skillful utilization* is very reliant on the model to adapt to any device and system. This expectation is unrealistic and, frustratingly, the model alone can only utilize a fraction of the achievable performance.

This situation leads programmers to *misuse* the programming model. A deep understanding of a particular device architecture is used to steer the optimization process. In extreme cases, programmers purposefully violate the model and rely on specific hardware characteristics to avoid undefined behavior, breaking execution portability – which is arguably an *abuse* of the model. This approach considerably deteriorates productivity and destroys portability. Sadly, it is necessary to achieve the best possible performance.

PRODUCTIVITY AND PERFORMANCE CHALLENGES The different usages of portable models described above are far apart on both the productivity and the performance spectrum. Yet, they are all using the exact same tool. Arguably, none of these approaches is ideal, since they are all based on a tradeoff between productivity *or* performance.

Improving productivity *and* performance together is necessary to enhance programmability of heterogeneous systems. To this end, automated and portable techniques are critical to exploit these models more efficiently.

1.3 CONTRIBUTIONS

Despite breakthroughs in the development of portable programming models, heterogeneous programming remains extremely complex. The previous section established that productivity and performance are in conflict. This paradox leaves heterogeneous systems largely underutilized and causes significant waste of human or computing resources. This thesis provides solutions to reconcile productivity and performance within a portable programming model. The main contributions are summarized below.

First, we demonstrate that a more efficient use of the model – specifically, OpenCL – is achievable transparently and systematically. By implementing the optimizations within the model itself, this solution is non-intrusive and can be deployed over any application.

Using profiling and compiler analysis, our runtime system can dynamically optimize a set of tasks delegated to an accelerator. Transformations like task parallelization, fusion and reordering – which are programmatically delicate and intrusive – are applied dynamically. This approach dramatically improves productivity and allows programmers to write modular, reusable code and reason about sequential task execution without having to give up any performance. We validate our technique by showing that the self-optimizing model not only replaces expensive manual code transformation but also outperforms human experts in dynamic and complex applications. This work is presented in Chapter 4.

Second, we developed new portable techniques to tap into hardware specific optimizations. Striving for performance above all, programmers painstakingly incorporate hardware knowledge into their application, destroying portability and productivity along the way.

We show that the same results can be achieved in an automated and portable fashion. Even though they are not expressible directly in the model, artifacts of the underlying hardware have noticeable side effects. These observations are synthesized to guide adaptive optimization strategies. We demonstrate this on a specific domain: stencil computations.

We show that PCI buses introduce heterogeneity in a seemingly homogeneous system. While this information is not programmatically available, we can implement an autotuner discovering and exploiting it. The data placement and communication patterns are optimized to reduce contention for shared resources, without requiring any explicit hardware information. These techniques are described in Chapter 5.

1.4 THESIS OUTLINE

This thesis is organized as follows:

CHAPTER 2 describes the state of the art in parallel programming and heterogeneous systems, in terms of the hardware architectures involved and their programming models. It focuses on GPUs, which are a relatively new type of compute architecture largely contributing to the heterogeneity of modern systems.

CHAPTER 3 provides an overview of prior work on abstraction and optimization techniques for heterogeneous systems. Works relevant to the two chapters below are then listed. First, the areas of JIT compilation and dynamic optimizations are discussed. Second, work related to high performance stencil computations is explored.

CHAPTER 4 describes transparent optimization techniques for the OpenCL model. A dynamic optimizer, named HELIUM, intercepts the Application Programming Interface (API) calls before they reach the vendor implementation. Through a mechanism of delay-optimize-replay, the commands are postponed and collectively optimized. Aggressive transformations, guided by a combination of dynamic profiling and static code analysis, replace considerable human efforts. The new technique is shown to match and even outperform manual optimizations on a set of benchmarks. This chapter is based partially on the work published in [LG14] and [LFC15].

CHAPTER 5 investigates portable autotuning techniques implementing hardware sensitive optimizations. Specifically, these techniques analyze and mitigate the impact of PCIe topology on communication in a multi-device setup. This approach is demonstrated by implementing PARTANS, a framework for high-performance distributed stencil computation. The integrated optimizer automatically adjusts for hardware artifacts, which are not programmatically available, without any explicit knowledge about the hardware. The findings of this chapter have been presented in [LFC13].

CHAPTER 6 concludes with a summary of the main findings from the new optimizations described above. The contributions are critically analyzed, and possible future extensions are discussed.

1.5 SUMMARY

This chapter introduced heterogeneous systems and their origins. Their inherent complexity requires a completely new approach to software development. Maximizing performance and productivity while preserving portability is a great challenge which calls for innovative optimization techniques. The contributions listed in this chapter introduced new solutions towards this goal. The outline of the thesis was provided.

The next chapter provides more detailed background information necessary to understand the problems and appreciate the contributions exposed in this thesis.

2 BACKGROUND

This chapter presents the background of parallel and heterogeneous computing, and introduces the technical terms and specifications necessary to understand the contributions detailed in the following chapters.

Section 2.1 provides a quick survey and a brief history of the most common components present in heterogeneous systems. Section 2.2 compares the architectures of CPUs and GPUs, which are the two main device categories used in our research. Their respective strengths and weaknesses are highlighted in the same section. Section 2.3 presents the main parallel programming concepts created to facilitate software development on heterogeneous systems. In particular, OpenCL is the framework used in our work. It is described in more detail in Section 2.4. Section 2.5 introduces the emerging trend of low level languages for heterogeneous systems, which is a key element for some of the work presented in Chapter 4. Finally, stencil computation, an application domain well suited for heterogeneous computing and studied in Chapter 5, is detailed in Section 2.6.

2.1 EVOLUTION OF PARALLEL COMPUTER SYSTEMS

This section provides a brief history of parallel systems and the evolution of CPU and GPU architectures. It then depicts a broader view of the heterogeneous computing landscape by mentioning other types of prevalent devices.

2.1.1 *From Single to Multi to Many Cores*

Chapter 1 introduced multi-core processors as a response from manufacturers to the end of Dennard scaling despite the continuity of Moore's law. The original multicore chip was introduced by IBM in 2001 and was meant for the high end server ranges. Intel then developed the first dual core for home use, the Pentium Extreme Edition, in early April 2005. AMD followed two weeks later with the release of two products: the Opteron 800 Series and Athlon X2 processors.

Since then, multi-core processors have completely replaced single core chips in all areas of computing, from high performance to embedded systems. They are also a powerful marketing tool in the competition between the main processor manufacturers.

Today, most commodity CPUs have four to eight cores. Their number is still increasing: the newest Intel Xeon processors [Int14] released in Q4 2014 provide between six and eighteen cores at a frequency of up to 3.40 GHz.

2.1.2 *General-Purpose Computing on GPUs*

Graphics processing has always been an important duty of computers. More elaborated user interfaces, quickly followed by fast growing game and scientific visualization industries, meant that a non negligible part of computation time was spent in rendering graphics.

Specialized hardware components were added to computers as early as the late eighties to lighten the rendering load: the Video Graphics Arrays (VGA) controllers. They started as simple memory and display controllers coupled together to do basic image transformations and rendering the output on monitors. VGA controllers then gradually increased in complexity, adding richer features and more compute power at each generation. In 1999, Nvidia released a new product, the *GeForce 256*, and for the first time used the term *Graphics Processing Unit* [NVI99].

Despite their rapid evolution, GPU devices remained very domain specific. For a long time, they could only handle a very restricted set of tasks, mainly rasterization and rendering-related operations like texture mapping, transformations and lighting. This was reflected in the architecture of these chips, which contained separate vertex units and pixel processors. However, the research community started to investigate the usability of these chips for general purpose computation in 2002 [Pur+02]. New programming models were adapted to the streaming processors used in GPUs as early as 2004 [Buc+04]. This marked the beginning of *General-Purpose computing* on GPUs (GPGPU).

In 2006, Nvidia released the GeForce 8800, the first GPU with a unified architecture and the same Instruction Set Architecture (ISA) for the vertex and pixel processors. The chip only has 128 stream processors – compared to thousands on the current generation – but its architecture is still fundamentally the same as modern GPUs.

More recently, a new type of processor has emerged: Accelerated Processing Units (APUs). APUs are hybrid chips containing a CPU and an accelerator on the same die. The embedded accelerator can be either a GPU, a Field-Programmable Gate Array (FPGA) or any other type of specialized processor. The goal of this design is ultimately to improve data locality by sharing the main memory or even some caches. All main GPU manufacturers are considering APU design: AMD provides several generations of APUs, Intel embeds a GPU in its Intel HD Graphics chips and Nvidia is developing a similar project codenamed *Denver* [Den].

2.1.3 *Other Parallel Systems and Accelerators*

While our research focuses primarily on systems composed of CPUs and GPUs, it is worth nothing that the heterogeneous system landscape is far more diverse.

In 2010, Intel released a new family of Many Integrated Core (MIC) products targeted to the HPC community, the Xeon Phi [Phi]. It provides over fifty cores with an X86 architecture and supports over a hundred concurrent threads. It is based on the Larrabee microarchitecure and – like GPU chips – it has a very wide vector unit (16 single precision floats). However, it remains a general-purpose processor and runs an operating system. The design of the cores is simpler than current CPUs. For example the first generations do not support out of order execution, but it is fully cache coherent.

FPGAs are integrated circuits containing programmable logic. Because of the difficulty in producing efficient hardware description languages and the complexity of software synthesis, they were traditionally used for hardware prototyping. However, recent advances in compiler technology and programming models for heterogeneous systems, like OpenCL, enables specialized frameworks to generate general-purpose code for FPGAs.

Finally, computation can be distributed amongst a large number of interconnected computers. Scheduling and synchronization are enforced through message passing protocols. This is how the first large-scale parallel machines were created, sparking a need for parallel programming. Today's largest supercomputers are still made of clusters of nodes, each composed of CPUs and accelerators [Top14]. Distributed computing is not explored in more detail in the remaining chapters: the techniques presented focus on optimizing single CPU systems. Distributing the computation to several nodes is an independent problem complementary to this research.

2.2 COMPARISON OF CPU AND GPU ARCHITECTURES

GPUs are prevalent in heterogeneous computing. Their relative performance, cost and power efficiency compared to CPUs allowed them to be present in all niches of computer science, from embedded systems to the largest supercomputers. This section describes their architecture in more detail and highlights the pitfalls of GPU programming.

Section 2.2.1 compares the architectures of CPUs and GPUs and Section 2.2.2 discusses their memory models. Finally, Section 2.2.3 enumerates their optimization challenges.

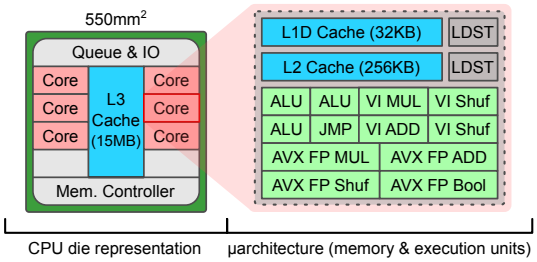
2.2.1 *Architecture Overview*

CPUs and GPUs have radically different architecture due to their different application domains. On the one hand, CPUs must be optimized for latency – they have to run multiple applications and the operating system smoothly. On the other hand, GPUs are optimized for throughput – they are used primarily for simple data parallel tasks such as image processing. This section discusses how these two strategies are reflected in their architecture.

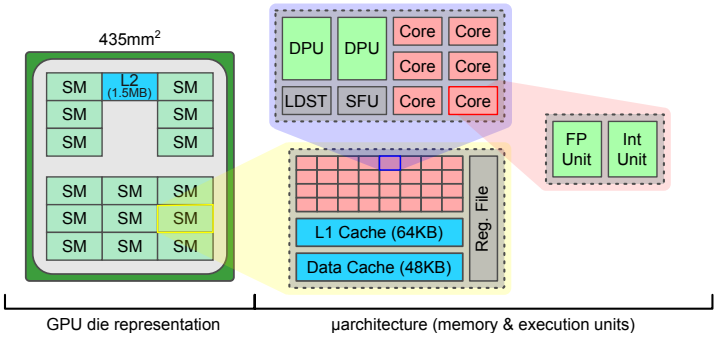
While their description uses a similar terminology, oftentimes the underlying hardware implementation and the purpose of the elements sharing the same name are completely different. These misnomers often lead to unfair comparisons. For example, the number of cores is several orders of magnitude greater on GPUs, typically thousands versus two to six. However, the functionality and design of these cores are fundamentally different.

On a CPU chip, most of the surface area is reserved for caches, controllers and sophisticated logic, as shown on the die representation in Figure 2.1a. Each CPU core has its own cache and a complex pipelined execution mechanism. They are able to reorder independent operations on the fly (out-of-order execution), or to evaluate execution paths ahead of time in order to keep the deep pipelines filled (branch prediction). This gives the CPU great flexibility for multi-tasking and general-purpose applications.

GPU chips pack thousands of much simpler cores, which are primarily intended for simple compute tasks. This results in a very high number of execution units at the expense of sophistication, as seen on the GPU die of the Kepler architecture in Figure 2.1b. The GPU "cores" have no register, no cache, no complex execution engine and no out-of-order execution. On Kepler, the GPU chip is divided into 15 Streaming Multi-Processors (SMX).



(a) Intel Sandy Bridge-E (i7-3960X) Architecture



(b) Nvidia Kepler (GK110) Architecture

Figure 2.1: Comparison of CPU and GPU. A larger area of the GPU die is dedicated to execution units compared to CPUs. CPUs have a lower number of cores, here 6 compared to 3840, but their cores have a more sophisticated design.

Each SMX has 192 single precision CUDA cores and 64 double precision CUDA cores. The SMX also holds the caches and most of the execution logic, instead of the core.

This design reflects the principal difference between CPUs and GPUs: their implementation of different execution programming models. CPUs exploit Multiple Instruction, Multiple Data (MIMD) and Single Instruction, Multiple Data (SIMD) parallelism while GPUs use a Single Instruction, Multiple Threads (SIMT) model. These models are represented in Figure 2.2.

MIMD is the most widely used technique to achieve parallelism on multi-core systems. There are a variety of sub-classifications for MIMD machines, but the common key idea is that independent processors, cores or execution units function asynchronously, independently and in parallel. At any time, several units of data are processed by potentially distinct instructions. In the case of multi-core processors, the different types of MIMD machines are distinguished by the underlying memory models. The principal classes are UMA (Uniform Memory Access), COMA (Cache Only Memory Access) and NUMA (Non-Uniform Memory Access).

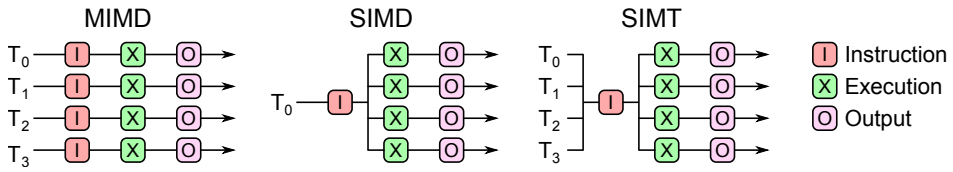


Figure 2.2: Concurrency models exploited by CPUs and GPUs. CPUs use a combination of MIMD and SIMD: each thread executes different instructions on data vectors. GPUs use a SIMT model: threads are grouped in batches executing the same instruction within a group.

SIMD is an execution strategy where one instruction is performed on a vector of operands at a time. Originally designed for supercomputers, vector processor units in CPUs have been around since the seventies. The concept is to maximize the throughput by decoding a single instruction, but executing it on multiple ALUs at the same time. Several data elements are then processed at once. Dedicated accelerators have used a similar design since their early days. For example, the Intel i750 video processor, released in 1990, could store two 8 bit values in one 16 bit register and perform the same operation on both in parallel.

In 1996, this technology was integrated to the Pentium MMX, which added vector integer operations on 64 bit wide registers. AMD then introduced floating-point operations support in 1998 with 3DNow to accelerate graphics rendering. Intel increased the width of the vector to 128 bits in 1999 with a new instruction set, called SSE, which was first integrated in the Pentium 3. The latest generation of Intel processors integrate a newer technology, AVX, which has up to 512-bit wide registers and can operate on 8 single precision or 4 double precision float values in parallel.

Modern multi-core CPUs exploit Data Level Parallelism (DLP) and Thread Level Parallelism (TLP) by combining MIMD and SIMD. They also implement Instruction Level Parallelism (ILP) via pipelining, branch prediction, speculation and other methods.

By contrast, GPUs have a hybrid execution model, called SIMT [1]. It behaves like a MIMD programming model combined with SIMD hardware. Threads are grouped in batches: each batch can process a different instruction, but all threads within a batch execute the same instruction in lockstep. Binary programs for GPUs are scalar; however, their execution is implicitly SIMD. Nvidia terminology defines a block as a *warp* which has a fixed size of 32 threads in all of their architectures to date. AMD defines a block as a *wavefront*, where 64

1 Not all GPUs use the SIMT model. For example, ARM's Mali GPU uses barrel threaded execution and Intel integrated graphics support multiple models. Larger dedicated GPU chips are at present all using the SIMT model.

	Size (KB)		Size/Thread[1] (KB)		Bandwidth (GB/s)	
	CPU	GPU	CPU	GPU	CPU	GPU
L1	6x64	15x64	32	8 Bytes	50-100	2.5TB/s
L2	6x256	1,536	128	50 Bytes	30-50	1KB/cycle
L3	15,360	n/a	1,280	n/a	20-30	n/a
DRAM[2]	1-24GB	6GB	>1GB	2184.5	30-50	200-300

Note: 1 – considering the total number of in-flight threads supported by the hardware
2 – on a system with DDR3 1,600MHz and GDDR5 7GHz

Table 2.1: On-chip and off-chip memory statistics for the i7-3960X CPU and the GK110 GPU.

threads operate in lockstep. The remainder of this section describes an Nvidia GPU as an example of GPU architecture and uses the Nvidia terminology.

2.2.2 *Memory Layout*

The memory structure and availability are also very important differences between CPUs and GPUs. In both cases, the memory is composed of small local caches, larger ones shared amongst threads and the main off-chip memory. While the hierarchy is roughly the same, the hardware implementation gives each layer very different features for both device types. Similarly to the compute cores, the memory is optimized for latency on CPUs and for throughput on GPUs. This leads to different – and sometimes opposite – optimization strategies. Table 2.1 summarizes the main characteristics of these caches for both devices.

2.2.2.1 *Main Memory Description*

The main memory, also called primary memory, is a critical resource for manipulating large data structures. Most applications require more storage space than the registers and caches can provide. The main memory provides this extra space, at a cost of lower bandwidth. However, this memory serves different purposes on CPUs and GPUs. Again, their hardware implementations differ.

CPUs usually use Double Data Rate (DDR) synchronous dynamic random-access memory as their primary memory. It is a very modular solution, allowing the amount of main memory to vary between a few gigabytes to several dozens. DDR3 has a peak transfer rate of 12 GB/s per module. However, this bandwidth does not scale linearly with the number of modules in the system. The principal limiting factor is number of channels and the

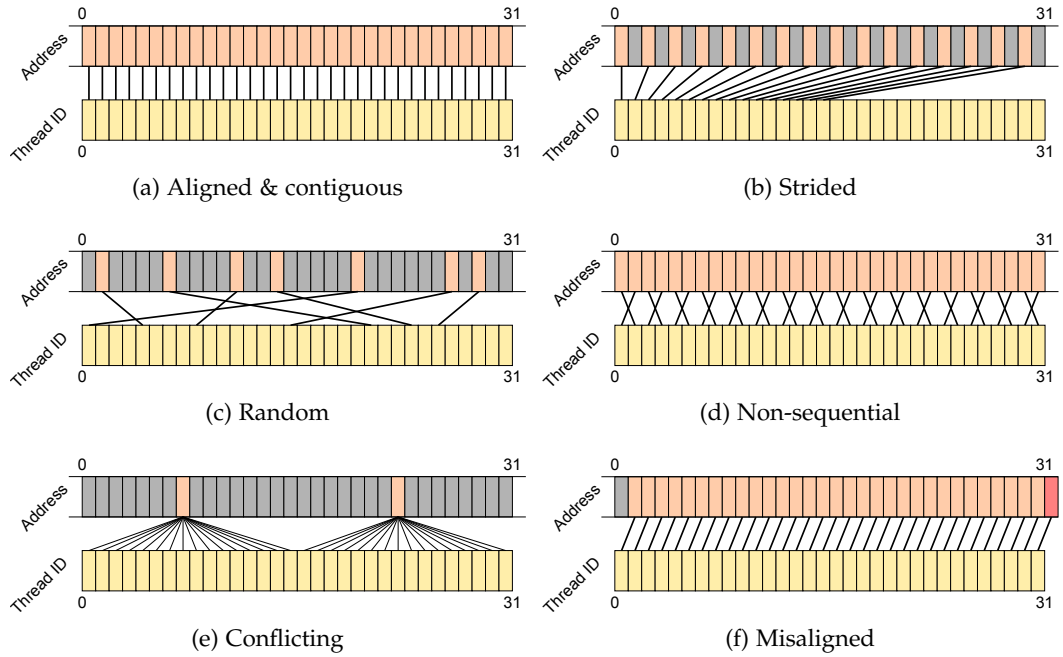


Figure 2.3: Examples of memory access Patterns. Each thread within a warp issues a memory request which maps a memory address. The memory addresses in these figures can either be an address in memory or a bank number.

bandwidth of the main bus. It varies on different systems and configurations, but the two most commonly used technologies – Hypertransport and Intel's Quickpath – have a main bus bandwidth limit of around 25 GB/s.

GPUs use synchronous Graphics Double Data Rate (GDDR) random access memory, which provides a high bandwidth at the expense of latency. The transfer rate for individual GDDR5 modules is 48 GB/s and the total main memory bandwidth for GPUs is about an order of magnitude higher than the CPU memory – with data rate around 230 GB/s. However, peak bandwidth is only achievable under very specific circumstances.

The strategy used by GPUs to amortize the memory latency and maximize the bandwidth is based on two factors. First, a large number of threads should be running in parallel to generate a sufficient amount of memory requests. Second, these threads should access memory in a very specific pattern. This last point is often the limiting factor since GPUs typically deal with large problem sizes – at least a few thousand threads – so they have enough concurrency. Hence a key optimization is improving memory access patterns.

This requirement is explained by the architectural features of graphics memory. Each memory bank is 128 bytes – or 32 words –, which corresponds to the granularity of a fetch operation. On Nvidia GPUs, memory requests are issued at a warp level. Hence, the total

bandwidth – provided enough requests are generated – is limited by the amount of words in a bank used by all threads in a warp. Optimizing this ratio is a process known as *memory coalescing*.

Figure 2.3 illustrates some examples of coalesced and uncoalesced memory access patterns. When every thread within a warp accesses a perfectly aligned and contiguous memory space (see Figure 2.3a), a single bank is fetched. This pattern gives the maximum possible throughput since every byte in the bank is used, and all the requests from the warp are satisfied at once. If the access is contiguous but falls between two banks (see Figure 2.3f), two transactions are necessary. Only 128 bytes will be used but 256 bytes are fetched, effectively halving the useful bandwidth. When the accesses are larger than a word or strided (see Figure 2.3b), the utilization of bandwidth decreases as the number of banks increases. For example, if the stride is two, every other four byte word in a bank will not be used. The number of memory transactions is then multiplied by two. If the gap between each address is larger than 128 bytes, there are as many transactions as requests, and the utilization drops to three percent – one word per bank. This also occurs when the addresses are scattered in memory (see Figure 2.3c).

Some devices provide hardware support for other types of operations such as broadcast for conflicting accesses (see Figure 2.3e), or shuffle for non-sequential patterns (Figure 2.3d).

2.2.2.2 *Cache Hierarchies*

Data caches hold the temporary runtime data necessary for computation. They are organized in a hierarchy: typically the closer they are to the core, the faster but also smaller they are. When a memory request is issued, it is redirected to the closest data cache. If the cache holds a valid and coherent reply to the request, the data is sent and the memory request is satisfied, this is a *cache hit*. If the data is not present, it is a *cache miss*; the request is forwarded to the next level in the hierarchy, until it is satisfied.

Most current CPUs have three levels of fully coherent on-chip caches. Their names reflect their conceptual distance to the core. L1 is a very low latency cache composed of an instruction and a data cache. It is designed to maximize the hit rate. L2 is a larger cache shared between several L1 caches. It is designed to minimize miss penalty. On the i7 3960x presented in Figure 2.1, both L1 and L2 cache are private to each core, and have a size of 64 KB and 256 KB respectively. The L3 cache is a much larger cache shared amongst all

cores. Its purpose varies depending on the chip, but its most common usage is to reduce delays in multi-threaded environments.

GPUs on the other hand do not have any caches or registers physically present in each core; they are instead located on each streaming processor. Modern Nvidia GPUs have a single non-coherent L1 cache per SMX and some data cache. The L2 cache is coherent and shared amongst all SMXs. While its size is significantly smaller than a CPU' L2 or L3 cache, its bandwidth is much higher. The Kepler architecture presented in Figure 2.1 packs 15 SMXs, each having 64 KB of configurable cache, all sharing 1.5 MB of off-chip L2 cache.

If one compares the size of the available caches and the amount of parallelism offered by each device, there is a striking difference in terms of resources per thread. Table 2.1 shows the distribution for the i7-3960X CPU and the Kepler GK110 GPU, whose architectures are compared in Figure 2.1. On the CPU, each thread has hundreds of kilobytes of available cache while on the GPU the smaller caches and the higher number of threads shrinks the size per thread to a couple of bytes. While this measure has little meaningful impact in practice, it highlights the contrasting execution strategies and the different roles played by memory caches on the two devices.

The reason why GPUs do not need a large cache per core lies in the way they handle memory requests. CPU caches are implemented to offer multiple advantageous cache effects to boost performance in a multi-tasking, multi-threaded environment. GPUs, on the other hand, are optimized for regular accesses in a lockstep execution environment. The GPU caches use the same bank size as the main memory. If the memory requests are coalesced, the number of transactions can be minimized with a large cache line, and all the bytes of the cache line will be used. For this reason, the L1 cache is used only to satisfy coalesced transactions from a single request originating from multiple threads from one warp. By contrast, the L1 cache on the CPU is used as a scratchpad space to optimize transactions temporally within and across threads. Thus, the caches have to be larger to increase the probably of a hit, and the cache line granularity has to be smaller.

2.2.2.3 *PCIe Interconnect*

While the Peripheral Component Interconnect Express bus (PCIe) is not directly part of the memory for neither the CPU nor the GPU, it plays an important communication role between the two. PCIe is currently the de facto standard to connect expansion cards, such

as GPUs, to the CPU and memory in a system. The bandwidth of these buses depends on many factors. Unlike its predecessor PCI, PCIe is not a shared parallel bus architecture. Instead, it uses a star-shaped topology with point-to-point links that connect devices to the PCIe root complex. Each link is composed from 1 to 32 full duplex PCIe lanes, determining the bandwidth of the link or slot. The number of lanes is usually given as a factor when one refers to the slot, e.g., PCIe x8 slot or PCIe x16 slot. However, this is not the only information required to reason about the available bandwidth to communicate with a GPU. Most systems have multiple PCIe root complexes. For example, one is embedded in the chipset, and another one is integrated to the I/O hub of the processor. These hubs are then linked by other means. QPI (Quick Path Interconnect) and DMI (Direct Media Interface) are some of the most common technologies. Motherboard and graphics card manufactures might use PCIe multiplexers or switches to increase the number of devices that can be physically connected to one PCIe link. The overhead introduced by these multiplexers is poorly understood, and their specifications are not available.

2.2.3 *Optimization Challenges on GPUs*

Optimization strategies for GPU applications are very different from those of multi-threaded programming. Unspecialized developers are not familiar with them, and they are not necessarily intuitive. This disparity is caused by the radically different execution models and memory mechanisms between devices found in a heterogeneous system. This section explains the main performance pitfalls and optimization strategies of GPU programming.

2.2.3.1 *Balancing Computation and Memory Bandwidth*

Because of its scale and execution model, high utilization is much harder to obtain on a GPU compared to a CPU. To maximize the efficiency of the device, computation and memory transactions have to be carefully balanced, which is not achievable for all applications.

For example, each SMX of a Kepler GPU can support 1,700 single precision math operations before saturating the instruction bandwidth, and the memory bandwidth saturates at 100 transactions of 128 byte lines per SMX. Problems are *compute bound* when they saturate the instruction bandwidth and under-utilize the memory bandwidth. One the other hand, a problem is *memory bound* if the number of requests saturate the memory bandwidth, in

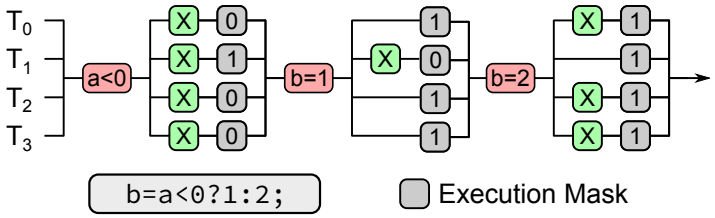


Figure 2.4: SIMT model and divergent code. Since threads operate in lockstep, divergent regions of code have to be evaluated sequentially, introducing a waste of compute resources. An execution mask is set at the beginning of the divergent region, holding the evaluation condition for each thread. The mask is then negated to evaluate the alternate path and reset after convergence.

which case compute resources are wasted. Uncoalesced memory accesses easily saturate the bandwidth with a high number of threads. For this reason memory bound problems are more common than compute bound problems on dedicated GPUs.

Scalability is also impaired by limited resources threads have to share, which define the *occupancy* of the device. Registers and local memory, for example, are resources shared by all threads on one SMX. If each thread requires a large amount of registers, or if a thread group requires a large amount of shared memory, the number of in-flight threads or concurrent groups will decrease to satisfy the dependencies. This decreases the amount of parallelism and makes it harder to reach peak performance.

2.2.3.2 *Thread Synchronization*

Parallel programming inherently introduces synchronization and collaboration issues between the concurrent compute units. Critical shared resources are exposed to races and have to be protected with locks and other exclusion mechanisms in order to ensure correctness. Fine-grained synchronization – like independent threads – makes concurrency management relatively easy on multi-threaded CPU applications.

This is however more complicated on GPUs using the SIMT execution model, since all threads in a block execute in lockstep, synchronization primitives are implemented in such a way that all threads in the block must take part. Adding a barrier in a divergent block of code results in a deadlock. Furthermore, there is no hardware support for synchronization across all threads in the device, making the implementation of some algorithms and data structures impossible on the GPU.

2.2.3.3 *Divergent Code*

Another impact of the lockstep execution is the overhead introduced by divergent code. Figure 2.4 shows an example of ternary statement executed by a SIMT model. Because they operate in lockstep, threads cannot independently execute both paths in parallel and converge afterwards. Instead, every thread in a warp executes all control flow paths taken by any other thread. In the example presented in Figure 2.4, all threads first evaluate the condition. Since at least one thread satisfies the predicate, the *then* branch is taken. Again, this is done by all threads. Because the control flow is divergent, execution masks are used to flag active threads. Inactive threads will remain idle instead of evaluating the instructions. In a second step, since not all threads executed the *then* block, the *else* path must also be executed. The execution mask is negated, and the alternative path is evaluated in the same fashion. Finally, the execution mask is reset once the control flow converged, and all threads proceed to evaluate the next instruction.

This lockstep evaluation introduces significant waste of resources for highly divergent code. The control flow is essentially evaluated sequentially, which is why GPUs typically suffer very poor performance for this type of applications.

2.2.3.4 *Data Layout*

Neighboring GPU threads should access consecutive memory locations in order to maximize bandwidth. This might require the programmer to change the data layout in the entire application, or to implement additional transformation steps before offloading the computation to a GPU.

A classical example of layout transformation is array-of-structure (AoS) versus structure-of-arrays (SoA) layout when using type aggregates [SLH12]. As established earlier, AoS introduces padding between elements and creates uncoalesced memory accesses, which considerably degrades performance. SoA packs elements of the same type contiguously, which increases effective bandwidth when accessing a single field but introduces a stride between the fields.

Another limitation caused by data layout is poor performance when using irregular data structures. Again, random memory accesses impair coalescing and decreases the effective memory bandwidth. Because of this, graph based applications or applications using hash tables or lists are hard to efficiently port to GPUs.

Counter-intuitively, it might be beneficial on the GPU to increase the overall amount of operations to improve performance. This is the case, for example, in a matrix-matrix multiply application: rotating one of the input matrices improves coalescing during the multiply phase at the expense of two additional memory accesses per element and extra computation for index computation. Since it is a memory bound problem, the additional complexity is largely amortized by the resulting bandwidth gains.

2.2.3.5 *Distributed memory Overheads*

The host application has to explicitly exchange data back and forth with the device. This is typically done through the PCI bus for dedicated GPUs.

Even considering the maximum achievable bandwidth – 1 GB/s per lane for PCIe 3.0 –, it is considerably lower than the bandwidth of the global memory on most GPU devices, which entails non-negligible communication overheads. This might become a bottleneck and outweigh the performance gain achieved by the GPU. Optimizing communication patterns and preserving data locality on the device is a crucial part of GPU programming.

2.3 PARALLEL FRAMEWORKS AND LANGUAGES

Parallel programming was once a domain restricted to the elite of software developers. Only they possessed the extensive knowledge required to operate exotic architectures. Parallelism was the realm of the first supercomputers and extended to a small niche of high-performance computing. Since then, parallel programming has become mainstream and needs to be accessible to all programmers, expert or novice. New concepts and programming models had to be created to confront this challenge.

This section presents the main frameworks and language extensions developed to achieve better utilization of parallel systems. The relevant terminology and concepts are introduced. A brief survey of the most commonly used frameworks demonstrates the different approaches for developing applications on multi-core CPUs and accelerators.

2.3.1 *Parallel Programming Concepts*

Parallel programming is an extensively studied concept because of its importance in modern computing. This section provides a brief overview of the main categories of parallelism, programming idioms and classes of parallel programming abstractions.

PARALLELISM TAXONOMY Parallelism can be expressed in three main categories: task parallelism, data parallelism and hybrid parallelism. Task parallelism – also called functional decomposition – consists in dividing a program in a number of independent tasks, which can be executed in parallel. Data parallelism – or domain decomposition – splits data in distinct subsets on which the same task can be applied independently. Task and data parallelism can be combined to offer better scalability.

CONCURRENCY MODELS Parallel programs are categorized according to two concurrency models: *shared memory* (SM) and *message passing* (MP). In the SM model, data and states are shared amongst the different workers. In the MP model, however, each worker has its own private data and they communicate amongst themselves using a messaging protocol. Heterogeneous systems often combine SM and MP models; where devices are organized in a hierarchy and communicate using MP, but a finer grain parallelism on each device allows a pool of workers to run concurrently on a shared memory. For example, a CPU and a GPU often have distinct main memory, but both devices can run multiple threads, which share their respective memory.

ABSTRACTION LEVELS There are three main levels of abstractions for parallel programming frameworks: *high-level*, *reduced abstraction* and *low-level*. The highest level of abstraction is achieved when the programmer does not require any knowledge about parallelism because it is entirely inferred – typically by compilers or code translators. Reduced abstractions provide a simplified view of the parallel architectures. Programmers need to be aware of parallel programming pitfalls and need to state concurrency in their application explicitly and carefully. While enabling reasonable scaling, these approaches do not achieve peak performance because of their limited and simplified model. To achieve peak performance, it is necessary to have a deep understanding of the targeted architecture and to use a low-level programming model to take full advantage of all the architectural features available.

ALGORITHMIC SKELETONS Software engineering often involves solving recurring problems to meet similar requirements. A concept of algorithmic templates called design patterns has been developed to provide simple, efficient and robust solutions to these fixed problems. High-order templates were developed to incorporate a functional aspect in the patterns, called *algorithmic skeletons* [Col04]. They are used extensively in parallel programming as a solution to integrate scalability in the design phase while being very flexible. Some of them expose task parallelism, like *task farms*, *divide and conquer* or *pipelines*. Others expose data parallelism, like *map*, *reduce* or *scan*. Well structured programs using skeletons are much easier to optimize and to map to parallel and heterogeneous architectures since the semantics of the algorithm is decoupled from the implementation.

2.3.2 *Parallelism on Multi-Core CPUs*

The previous section introduced the three levels of abstraction: high level, reduced and low-level abstractions. This section lists the main frameworks in each of these categories to give an overview of how the parallel programming concepts are applied in practice on multi-core CPUs.

The highest level is automatic parallelization [SL05]. While a lot of research has been done on this field, it remains mainly an academic subject. Finding and exploiting a large amount of the available parallelism automatically is extremely difficult. A provably correct usage of critical resources without user intervention requires complex static analysis and must be very conservative about dependencies. Some compilers like Intel's C++ Compiler and GNU's C++ compiler have automatic parallelization features. Sadly, they are mainly limited to simple loops.

To achieve better performance, user input is necessary to hint parallelism to the compiler and provide guarantees which cannot be statically checked, like the absence of data race. This can be done in a non-intrusive way using compiler directives. OpenMP (Figure 2.5b) is an example of compiler directive framework allowing programmers to express both data and task parallelism.

```
void saxpy(int n,float a,float *x,float *y){
  for(int i=0; i<n; ++i)
    y[i] = a*x[i]+y[i];
}
```

(a) Sequential version

```
void saxpy(int n,float a,float *x,float *y){
  #pragma omp parallel for
  for(int i=0; i<n; ++i)
    y[i] = a*x[i]+y[i];
}
```

(b) OpenMP

```
void saxpy(int n,float a,float *x,float *y){
  vector<thread> pool; pool.reserve(n);
  for(int i=0; i<n; ++i)
    pool.emplace_back(thread{
      [=](){y[i] = a*x[i]+y[i];}
    });
  for(auto &t : pool) t.join();
}
```

(c) C++11

```
void saxpy(int n,float a,float *x,float *y){
  tbb::parallel_for(
    tbb::blocked_range<int>(0,n),
    [&](tbb::blocked_range<int> &r) {
      for(int i : r) y[i] = a*x[i]+y[i];
    }
  }
}
```

(d) TBB

Figure 2.5: Multi-threaded program chrestomathy for SAXPY(a). User annotations like OpenMP(b) guide compiler auto-parallelization. Threads are natively expressed in C++11(c). Many libraries, like Intel's TBB(d), provides high-level abstraction for complex tasking model.

Some languages expose concurrent structures and concepts as a core feature. This is especially the case of modern computing languages. These concepts are taking much longer to be integrated to older languages like C++ or Fortran. The concepts of threads and concurrency have been added to the C++ standard recently, in 2011.

A low-level API (Figure 2.5c) is provided and programmers still have to implement most of the tasking model.

High level APIs have been developed in an effort to abstract the complexity of the low-level details of concurrent programming. In addition to simplify the code, they often provide a much more advanced tasking model and tuned heuristics to improve performance. The most commonly used libraries are Apple's Grand Central Dispatch [SF12] and Intel's Threading Building Blocks [Rei07] (Figure 2.5d), which provide both task and data parallelism and a collection of algorithmic skeletons.

2.3.3 *Languages and Frameworks for Accelerators*

While most concurrency libraries and frameworks for multi-core CPU aim primarily to improve performance and productivity, tools for accelerators are also concerned with portability. Some programming paradigms focus on a single type of device. Alternatively, they

```
kernel
void saxpy(float a, float4 x<>, float4 y<>,
           out float4 res<>)
{ res = a * x + y; }
```

(a) Brook

```
__global__
void saxpy(int n,float a,float *x,float *y){
  const int i = blockIdx.x * blockDim.x +
                threadIdx.x;
  if (i < n) y[i] = a*x[i]+y[i];
}
```

(b) CUDA

```
kernel
void saxpy(float a, global float* x,
                    global float* y){
  const int i = get_global_id(0);
  y[i] = a*x[i]+y[i];
}
```

(c) OpenCL

```
void saxpy(int n,float a,float *x,float *y){
  #pragma acc kernels
  for(int i=0; i<n; ++i)
    y[i] = a*x[i]+y[i];
}
```

(d) OpenACC

```
thrust::device_vector x, y;
thrust::transform(
    x.begin(), x.end(),
    y.begin(), y.begin(),
    a * _1 + _2);
```

(e) CUDA/Thrust

```
vector<float> x, y;
bolt::transform(
    begin(x), end(x), begin(y), end(y),
    [=](float _x, float _y) restrict (cpu,amp)
    { return a*_x+_y; });
```

(f) C++AMP/Bolt

Figure 2.6: GPGPU programming is usually expressed using specific languages matching the execution model. These languages include Brook(a), CUDA(b) and OpenCL(c). In these models, the iteration space control (not shown) is decoupled from the compute kernels. Compiler frameworks, such as OpenAcc(d), provide non-intrusive support. Finally, high level libraries like Thrust(e) or Bolt(f) offer increased productivity.

factorize and generalize common properties of classes of hardware and propose a generic abstraction

Libraries and frameworks for accelerators take the same form as those for multi-core CPUs: compilers tools, abstraction libraries and languages. The first sound development kit for GPGPU programming was a framework called Brook [Buc+04], providing all three: a new language (Figure 2.6a), an abstraction API and a compiler. It defined the modern concept of streaming programming style and a new compute target: graphics processors.

In 2007, Nvidia released a software development kit for parallel computing on GPUs called Compute Unified Device Architecture (CUDA) [Cud]. It extended the programming model provided by Brook and was based on the same types of tools: a language (Figure 2.6b) and a compiler. Since then, CUDA has had an increasing number of features as GPU architectures are becoming more complex, but it remained restricted to Nvidia GPUs and does not support any other platform.

In an effort to improve code base portability across CPUs and GPUs, Apple, along with AMD, IBM, Intel, Qualcomm and Nvidia proposed a new approach called Open Compute Language (OpenCL). It was standardized by the industry consortium Khronos in 2008 [Gro08] and representatives from CPU, GPU, embedded-processor, and software companies were involved in its specifications [Opeb]. It defines a portable language (Figure 2.6c) – based closely on the programming model offered by CUDA. A dynamic runtime enables portability at a program binary level. The work described in this thesis is mainly based on the OpenCL framework, and its model is discussed in more detail in Section 2.4.

Just as TBB provided high level abstractions for multi threaded applications, many libraries have been implemented to improve the productivity of these new languages. Thrust [HB10] provides high level functions for data access patterns and operations (Figure 2.6e), based on an API similar to the C++ Standard Template Library. Bolt [HR10] provides a similar abstraction (Figure 2.6f) for C++AMP.

Finally, some approaches bring portability to existing languages. OpenACC [Opea] is a standard for compiler annotations in C++ (Figure 2.6d), which provide a non-intrusive syntax similar to OpenMP.

2.4 HETEROGENEOUS COMPUTING WITH OPENCL

OpenCL is the first widely adopted standard for heterogeneous computing. At last, the same code base can reliably compile and run on multi-core processors, graphics cards, FPGAs and other accelerators. An increasing number of hardware vendor provide OpenCL support for their devices, allowing its coverage to span even further. Since its first iteration in 2008, the standard went through three minor revisions and one major, each time improving portability and providing a richer set of features.

The programming model proposed by OpenCL and its core implementation has remained the same since the original version. Section 2.4.1 presents the execution model and explains how it is portable across many architectures. Sections 2.4.2 and 2.4.3 detail the API of the host and device code respectively. Finally Section 2.4.4 discusses the limitations of the OpenCL framework.

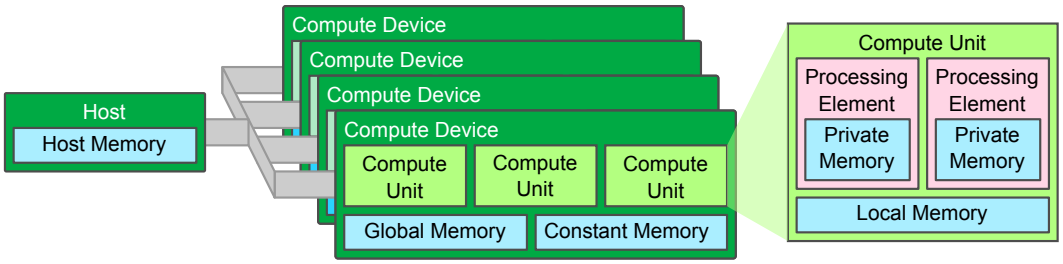


Figure 2.7: OpenCL Platform and Memory Abstract Model. A heterogeneous system is composed of devices, each has a set of compute units which can process multiple elements in parallel. Each component has its associated memory with a conceptual cost for each level.

2.4.1 *A Portable Programming Model*

The programming model of OpenCL is composed of conceptual representations for device architecture, memory model and computation. These generic concepts are then mapped to specific hardware in different ways, which is oblivious to the application built against the model. This is the key of the execution portability offered by OpenCL. The three concepts are detailed below.

PLATFORM MODEL OpenCL provides an abstract hardware architecture by aggregating properties common to multiple devices. Figure 2.7 shows the resulting *platform model*. The *host* application contains the non-OpenCL part of the program as well as the interaction with the model itself. It can query properties about the available *platforms*, which contain a list of *compute devices*. Each compute device has multiple independent *compute units* (CU), and each of those has local parallelism with multiple *processing elements* (PE).

This model can accommodate both CPU and GPU architectures. On a CPU, CUs can be mapped to cores, while on the GPU they can be mapped to stream processors. PEs can be thought of as a thread; they are independent workers which all run concurrently within a compute unit.

MEMORY MODEL The OpenCL model also provides an abstract memory hierarchy – which does not necessarily have to be implemented in hardware. The conceptual cost of accessing these memories increases the further they are from the PEs, but their conceptual size increases. Each PE contains some *private memory*, which is not accessible by other PEs. PEs on the same CU share some *local memory*, which is not accessible from outside the CU.

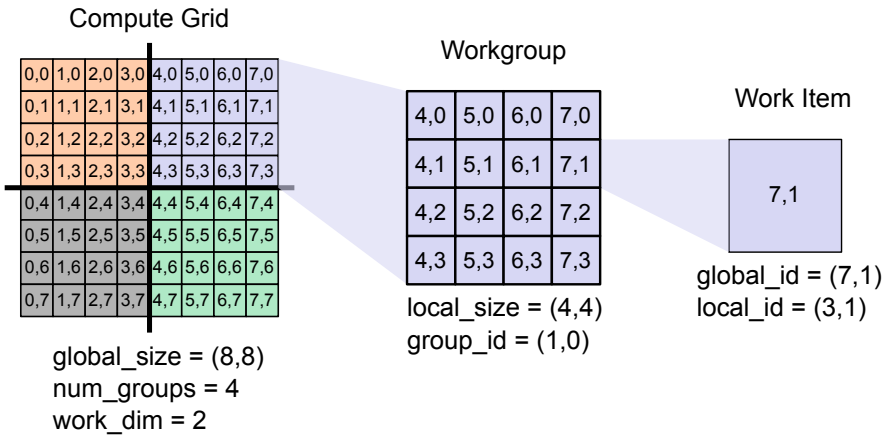


Figure 2.8: OpenCL execution model. Computation is expressed as functions mapped to a multi-dimensional domain called compute grid. This grid is divided in sub-domains called workgroups. Each level has a set of properties about the domain and the position of the element in the domain which can be queried though the API.

Finally, the device has some *global memory* and *constant memory*, which is shared amongst all the CUs of the device.

On the GPU, this maps well to the actual architecture. Private memory corresponds to register files, local memory is part of the L1 cache located on the SM, while global and constant memories are located on off-chip RAM. Some GPUs provide hardware support to improve the efficiency of the constant memory compared to the global memory.

On the other hand, CPUs do not have the same hardware hierarchy. Private memory is still mapped to register files but all other levels are mapped to the main memory. However, the same visibility and accessibility rules apply; and it is encouraged to use the abstract model to express data locality whenever possible. While there is no hardware advantage in doing this, the compiler might use semantic information from the OpenCL model to perform more aggressive optimizations and also improve performance on the CPU.

EXECUTION MODEL The OpenCL execution model, presented in Figure 2.8 explicitly exposes data parallelism. A function is instantiated for each point of a multi-dimensional domain called the *compute grid*. The domain is divided in sub-domains of equal size called *work groups*. Each element of the work group is a *work item*.

The model also provides guarantees in terms of parallelism and synchronization: all work items within a work group must execute in parallel, but the work groups themselves are not guaranteed to be processed concurrently. This means work items within a work

group can be synchronized; but multiple work groups, and by extension the entire grid, cannot be synchronized.

2.4.2 *Architecture Agnostic Compute Kernels*

The device code represents the core of an OpenCL program written in an OpenCL specific language. Recent versions of OpenCL support OpenCL-C and OpenCL-C++, which are based respectively on a limited subset of C99 and C++14 with some restrictions and some extensions.

A device program is composed of a list of function definitions. It may contain several entry points, which are functions annotated with the *kernel* keyword. The body of a kernel function is very similar to a C function, with a few limitations, which are mainly due to a lack of support for I/O, recursion and dynamic allocation.

However, the device languages provide an API for geometry processing, a comprehensive set of mathematical functions and support for vector types and operations. A kernel function is instantiated for each element of the compute domain, as explained in the OpenCL execution model description. The instance can query its position in the global index space and the local block shown in Figure 2.8 by using the API functions *get_global_id* and *get_local_id* respectively. It can also query the size of the index space, the size of a block and the block index of the current element.

A kernel instance is computed by a PE in the OpenCL platform model, while the local block is processed on a single CU. The abstract memory hierarchy of the model states that CUs can share local memory, which can be used from the device code by qualifying pointers as *local*. Similarly, global memory is qualified with the *global* keyword and constant memory with *constant*.

A kernel instance also corresponds to a work item in the execution model: instances within a work group can be synchronized using a *barrier* functions, but different work groups cannot be synchronized. Only termination of a kernel function guarantees synchronization across all work items in the domain.

Figure 2.6c shows an example of OpenCL-C code for SAXPY. The kernel function takes a scalar value and two global arrays as argument. The function accesses the arrays at the index of the current instance and performs computation for a single point in the domain.

SAXPY host code in Python

```
1  import numpy as np
2  import pyopencl as cl
3  # Input data
4  x_np = np.random.rand(50000).astype(np.float32)
5  y_np = np.random.rand(50000).astype(np.float32)
6  # Create context and queue
7  ctx = cl.create_some_context()
8  queue = cl.CommandQueue(ctx)
9  # Allocate device buffers
10 mf = cl.mem_flags
11 x_g = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=x_np)
12 y_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=y_np)
13 # Load and build  OpenCL-C program
14 prg = cl.Program(ctx, open("kernel.cl", "r").read()).build()
15 # Start kernel
16 prg.saxpy(queue, x_np.shape, None, np.float32(10), x_g, y_g)
17 # Read result
18 cl.enqueue_copy(queue, x_np, x_g)
```

Figure 2.9: Example of OpenCL host program in Python. The host code is in charge of allocating memory on the device (l11-12), dispatching computation (l16) and ensuring memory consistency between host and device (l18).

Finally, the device languages offer a collection of features for image processing and rendering. A built-in image type supports common image formats and an interoperability layer with OpenGL for rendering. These features are not used in this work and will not be described in detail.

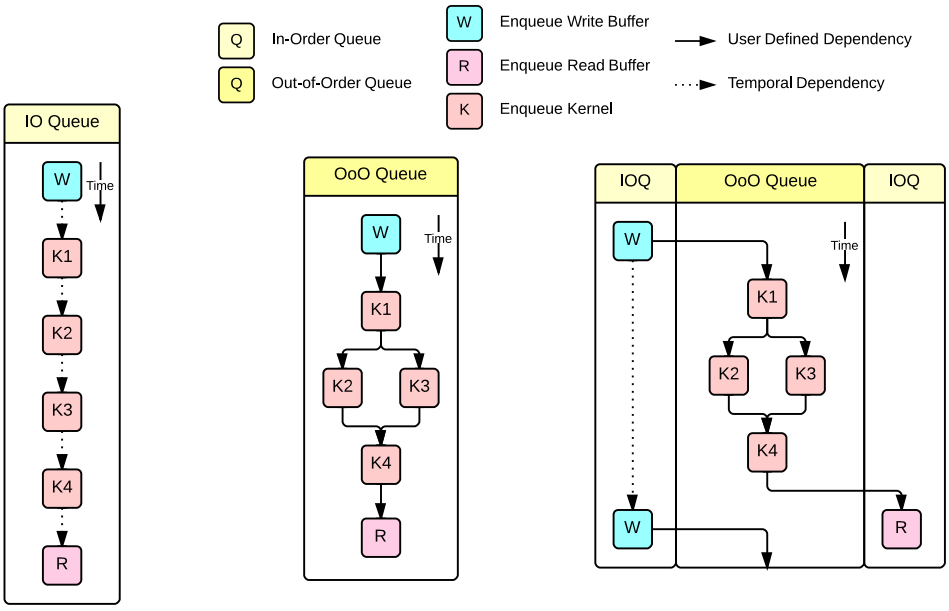
2.4.3 *Managing Data and Computation*

While the core computation is expressed in a separate device program, it is necessary to delegate computation to the appropriate accelerator and to manage the memory coherency of the data across host and device programs. This is the role of the *host application*.

The host application can be written in many different languages having an OpenCL binding, but the original OpenCL standard provides a list of functions as a C API. This API allows the application to list and set up devices at runtime in a platform independent way, thus ensuring portability of the application. Figure 2.9 is an example of host program written in Python showing the different components of a host application.

The host code is centered around *execution contexts*, which ensures consistency amongst the various OpenCL runtime objects. Those objects can be *devices*, *allocated memory* or *kernels*.

Memory *buffers* can be allocated and attached to a context. Often, these buffers mirror existing data structures of the host application, which are used to initialize data on the



(a) *Single In-Order Queue* (b) *Out-of-Order Queue* (c) *Multiple Queues*

Figure 2.10: Example of command queue strategies in OpenCL. 2.10a is the simplest model, where all the actions will be executed in the order they are enqueued. 2.10b exploits task parallelism, but it requires users to explicitly define the dependencies between tasks. 2.10c is an hybrid model where IO is processed in order and computation out of order.

device or manipulate data from the host side. The consistency between the host memory and device buffers is explicit: data is either written to or read from the device.

Device *programs* are also tied to a context. They contain a collection of OpenCL kernels used by the application. Programs can be JIT compiled at runtime from OpenCL-C/C++ source, and the resulting binary run on all the devices sharing the same execution context.

Kernel objects can be created from a program using the name of the kernel function. OpenCL-C kernels do not contain any information about the domain they will be applied to; this has to be specified in the host application. When starting a kernel, the API describes the shape of the grid and blocks, and an offset in this domain.

Memory communication and kernel launches are managed by *command queues*, which are specific to a single device. While OpenCL is used primarily for data parallel programs, its execution model supports both data parallelism and task parallelism, since most OpenCL actions are asynchronous. Data parallelism is expressed at a kernel level through grid processing, and task parallelism is expressed using command queues.

Task parallelism can be achieved by creating multiple queues. If the device supports concurrent tasks, they will be executed in parallel. Another way to express task parallelism is to use out-of-order queues instead of the default in-order queues. Out-of-order queues en-

able the runtime to re-order actions and execute them in parallel if possible (Figure 2.10b). While it does not guarantee parallelism, it relaxes the constraints of in-order queues, which behave as First-In First-Out queues and guarantee completion of a task before starting another (Figure 2.10a). In-order and out-of-order queues can be combined to define complex pipelines. Figure 2.10c shows an example of application using three command queues. The input and output data are fed sequentially in separate in-order queues to guarantee. Computation uses an out-of-order command queue to exploit task parallelism.

To avoid data races and incoherent results, conflicting tasks in multiple queues or out-of-order queues must be carefully scheduled in the host code. This can be expressed either using a dependency mechanism or synchronization primitives.

The dependency mechanism is expressed through *event* objects. Every action pushed in a command queue creates an event representing it. The command can also be given a list of events representing previously enqueued tasks. It can only start once all its dependencies have finished. This allows programmers to build complex task graphs of dependent actions to expose task parallelism while ensuring the coherency of the scheduling.

It is also possible to manage asynchronous tasks using synchronization primitives. They can be expressed either by launching a blocking action in a command queue or using an explicit wait for a list of events. Both will block the execution of the host program until completion of the operation. A synchronizing command might implicitly synchronize other pending operations, for example precedent tasks in an in-order queue. Therefore it is important to keep track of the synchronization points between the host and the device when moving data from one to the other in order to maintain coherency of the data.

Using a mix of task and data parallelism is complex. An incorrect scheduling is difficult to spot, since it does not generate an error in the program, but rather introduces wrong output. For this reason, very few programs take advantage of task parallelism.

2.4.4 *Limitations of the OpenCL Model*

While OpenCL provides a good hardware agnostic architecture, some aspects of the programming model still prevent programmers from writing high-performance code efficiently. This is because the optimizations required to achieve peak performance on a particular device are inherently architecture specific. Section 2.2.3 listed some of these differences in

the context of CPU versus GPU optimizations, but similar differences might exist between devices of the same type also.

An example of such optimization is choosing an appropriate data layout, as presented in Section 2.2.3.4. There are many ways of representing type aggregates in memory, either by changing their ordering, or changing their type altogether, and this can affect performance by an important factor. For instance, an efficient representation of matrices on the CPU is row major, because it improves intra-thread data locality. However, on the GPU, a column major representation, prioritizing inter-thread locality, allows adjacent threads in a group to access consecutive data in memory to maximize the bandwidth utilization. For applications using type aggregates, it is also common to switch between structures-of-arrays and arrays-of-structures depending on the type of computation being applied. For image processing, for example, a color image can be represented either with interleaved or independent channels.

Such optimizations have side effects beyond the compute kernels and require a substantial re-writing of the host program, introducing additional scatter-gather steps to shuffle the data, which might or might not be beneficial. Thus, these transformations are often not considered, and more effort is spent on optimizing computation kernels.

2.5 TOWARDS PORTABLE INTERMEDIATE LANGUAGES

Section 2.3.3 listed high level programming tools for heterogeneous systems, and Section 2.4 provided a more detailed description of the OpenCL framework, which is considered as a low-level programming model by the community. In this section, we describe even lower level languages, typically embedded in compiler frameworks, which also start to accommodate heterogeneous systems.

2.5.1 *Standardization of Intermediate Representations*

To improve portability, frameworks like OpenCL use just-in-time compilation of the code executed on the accelerator. However, this compilation has a non negligible cost, since all steps of the compilation have to be performed, from frontend to optimization to backend. To reduce the overheads, some lower level alternatives have been proposed.

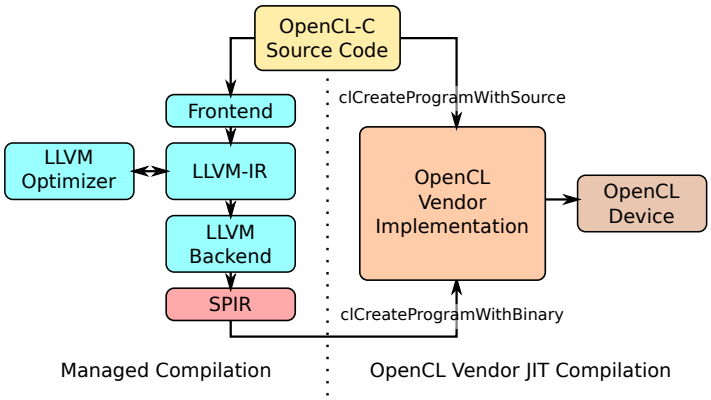


Figure 2.11: OpenCL device program compilation alternatives. OpenCL-C sources can either be loaded and compiled directly through the OpenCL API. Alternatively, users can implement their own compiler and load the program as a binary form using SPIR.

Nvidia has been using a low-level intermediate language for the CUDA framework, called Parallel Thread eXecution (PTX). This was an effort to improve compatibility across generations of devices with a slightly different Instruction Set Architecture (ISA). PTX is the target of the CUDA compiler, and a separate backend compiles PTX code to the actual target ISA.

In order to bring similar functionalities to OpenCL, the Khronos group introduced the Standard Portable Intermediate Representation (SPIR) specifications [Spi]. SPIR is a much lower level code than OpenCL-C, based on Low Level Virtual Machine (LLVM)'s intermediate representation. SPIR code can be loaded into an OpenCL implementation as a device binary program containing kernel functions. Most of the OpenCL vendor implementations are already based on LLVM, making the adoption of this new standard straightforward. However, it allows other languages and frameworks also based on LLVM to benefit from the OpenCL runtimes.

Another low-level language specification has been developed as part as AMD's Heterogeneous System Architecture (HSA) initiative: HSAIL. This provides a virtual ISA and an intermediate translation language which it mapped at runtime to a vendor ISA. Its goal is to provide a common ISA to allow existing languages to offset computation to multi-core CPUs, GPUs and other devices.

```
define spir_kernel void @saxpy(float %a,float addrspace(1)* nocapture %x,
                               float addrspace(1)* nocapture %y) {
entry:
  %call = tail call cc75 i32 @_Z13get_global_idj(i32 0) nounwind readnone
  %arrayidx = getelementptr inbounds float addrspace(1)* %x, i32 %call
  %0 = load float addrspace(1)* %arrayidx, align 4, !tbaa !8
  %arrayidx1 = getelementptr inbounds float addrspace(1)* %y, i32 %call
  %1 = load float addrspace(1)* %arrayidx1, align 4, !tbaa !8
  %2 = tail call float @llvm.fmuladd.f32(float %a, float %0, float %1)
  store float %2, float addrspace(1)* %arrayidx1, align 4, !tbaa !8
  ret void
}
```

Figure 2.12: SPIR representation of SAXPY. Using a low level representation enables compilers to decouple computation from high level languages and architecture specific backends.

2.5.2 *A Closer Look at LLVM and SPIR*

Low Level Virtual Machine (LLVM) is a compiler toolchain. It is combined with Clang to provide a three-phase compiler. Clang is a frontend for C based languages; it supports C, C++, Objective-C and OpenCL. The frontend generates an Intermediate Representation (IR), called LLVM-IR. LLVM provides a set of passes in the middleend to transform and optimize the IR. Finally, LLVM provides a collection of backends to generate binaries from the IR. The backends can be used as part of the compilation pipeline but also at runtime using a JIT compiler API. Amongst the many backends, LLVM can generate X86, PTX and Standard Portable Intermediate Representation (SPIR) code.

SPIR is a recent standardization of a subset of LLVM-IR. Similarly to PTX for CUDA, SPIR is a compiler oriented middleware representation providing a platform agnostic low-level code which can be compiled by a vendor specific compiler. SPIR can be used by any OpenCL 2.0 device – as opposed to Nvidia devices only for PTX. While it does not address the technical limitations of OpenCL discussed in Section 2.4.4, SPIR has several advantages and aims to widen the use of OpenCL. By providing a standard middleware, SPIR decouples the device code from OpenCL-C and allows other frontends to generate code usable by the OpenCL runtimes. An example of SPIR program is shown in Figure 2.12.

2.6 STENCIL COMPUTATIONS

Stencil computation [MRR12] is a frequent computational pattern present in many domains, such as image processing, algebra or continuous and discrete physics simulations. It is used

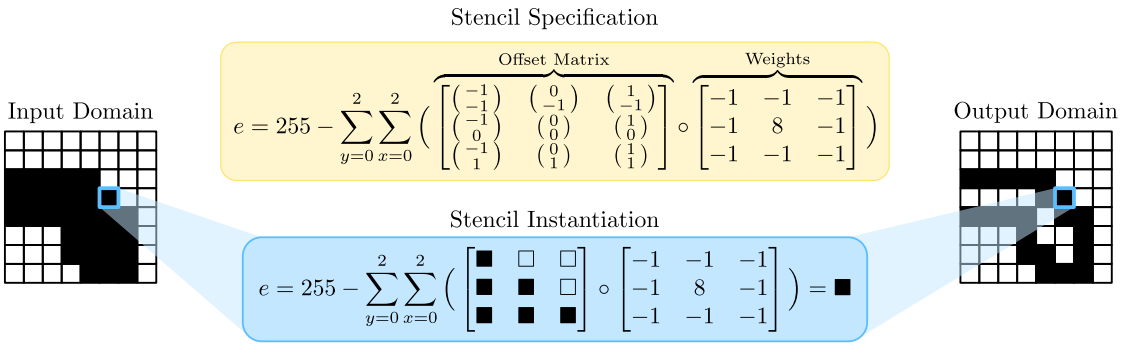


Figure 2.13: Example of stencil computation: simple Edge detection filter using a 9 point stencil. A function defined in the *stencil specification* is *instantiated* for each element of the *input domain* in to the *output domain*.

as a case study in Chapter 5 to explore hardware specific optimization techniques. This section introduces the properties and terminology of this compute pattern, and exposes the challenges of distributed stencil computation.

2.6.1 *Stencil Pattern*

Every point of a domain is updated, often iteratively, as a function of its surrounding elements. An example of a simple edge detection algorithm is shown in Figure 2.13. It can be defined by three characteristics: its *domain*, the *stencil operator* and the *boundary conditions*.

STENCIL DOMAIN The computation domain of a stencil, also called *volume*, can take many forms. For regular stencils, it is expressed as a multi-dimensional Cartesian grid, where the computation is carried out at each point. The dimensionality of the volume depends on the application: signal processing typically uses one-dimensional volumes, image processing two-dimensional images and physics simulations are often three or four dimensional.

STENCIL OPERATOR The stencil operator represents the point-wise computational pattern which is mapped to each element of the domain. It is characterized by an access pattern of adjacent elements, known as the *stencil shape*, and a function computing the value of the domain at the next time step.

Stencil shapes have very varied forms; they are usually defined by the number of neighboring points accessed. Figure 2.14 shows various examples of stencil operators, from a single-point computation without spatial dependency, to a 64-point 3D stencil.

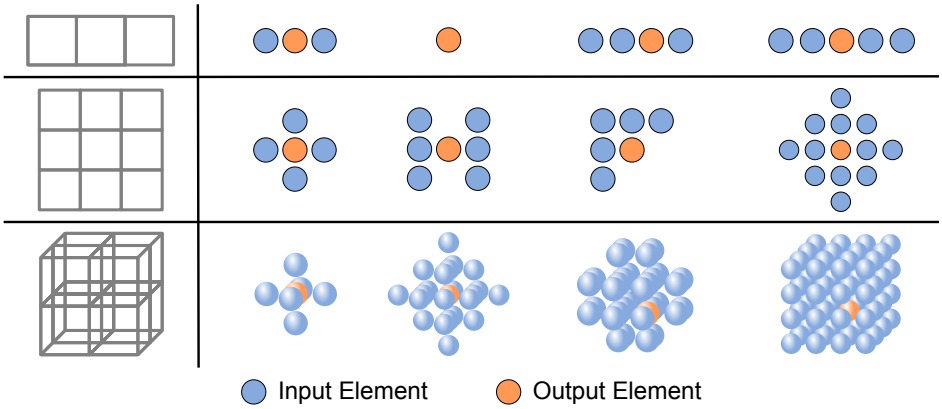


Figure 2.14: Example of stencil shapes in 1, 2 and 3 dimensions. Stencil shapes represent the location of the neighboring elements accessed from the stencil kernel with respect to the element being processed. The same access pattern is applied across the entire domain but it does not have to be regular or symmetrical.

The maximum offset in any dimension defines the *space-order* of the stencil. The operators from Figure 2.14 have space orders between 0 and 2. Similarly, the time dimension is associated to *time-order*. Operators reaching further than one timestep are called *high time-order* stencils.

BOUNDARY CONDITION The spatial dependencies of the operator cannot be satisfied at the boundaries of the domain. Some applications avoid this problem by only computing the inner parts of the domain, ignoring the edges. Others define a specialized behavior to dictate how the value of the neighbors falling outside the domain should be computed. These different strategies are called boundary conditions.

The most common, represented in Figure 2.15, are:

- *Mirror* and *bounce*: the distance vector outside the domain is reversed, respectively including and excluding the last elements. For multi-dimensional overflow, the vector can be reverted using by symmetry or antisymmetry.
- *Periodic wrap-around*: the distance overflowing the domain in each dimension is translated to the opposite side, creating an n-dimensional torus.
- *Clamp*: the elements outside the domain are ignored. This strategy often involves a specialized version of the stencil operator to account for the missing elements.
- *Fixed* and *interpolate*: use either a constant value or an interpolation function to compute elements outside the domain.
- *Extend*: the closest element from the domain is used.

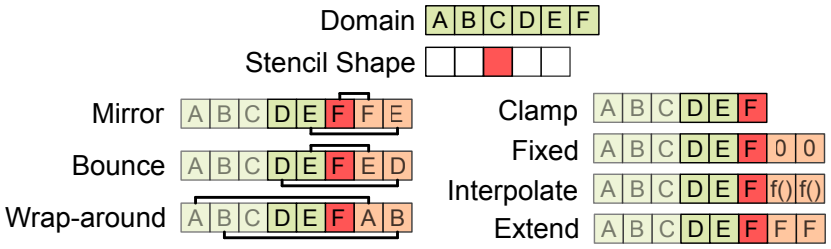


Figure 2.15: Examples of boundary condition. A five point stencil is applied on the edge of a 1D domain. The spatial dependencies outside the domain can be retrieved using different strategies, called boundary conditions.

2.6.2 *Distributed Stencils*

Since stencil patterns are very regular, computing time and memory traffic tend to grow rapidly and linearly with the number of elements. This dramatically impacts performance, especially for volumes with a high dimensionality. For this reason, and because stencil computation is a fundamental algorithm in many domains, parallel implementations of stencil computations are necessary to process large domains in a reasonable time frame.

Despite being embarrassingly parallel, stencil computations are not easy to map to a distributed memory model. Since each element has spatial dependencies to other elements of the volume, the domain cannot be split in independent sub-domains without breaking these dependencies at the boundaries.

To distribute a stencil computation, the sub-domains must overlap to satisfy spatial dependencies at the boundaries. This creates redundant parts of the domain called *halo regions*. The coherence of the halos has to be carefully maintained, which adds considerable complexity to the code. Furthermore, elements at the boundaries are loaded multiple times by different sub-domains, which introduces redundancies and impairs scalability.

For iterative stencils, extra complexity is introduced by the fact that there is also temporal dependency combined with spatial dependencies. At each iteration, the elements at the boundaries of the sub-domains cannot be computed, which shrinks the sub-domains after each timestep. In order to compute multiple timesteps, the amount of overlap can be increased.

Figure 2.15 shows an example of halo consumption with different stencil shapes for a halo size of 2. The stencil used at the right-hand side uses a single neighbor, but the tiles overlap by two elements, hence two timesteps can be computed. The same halo size with

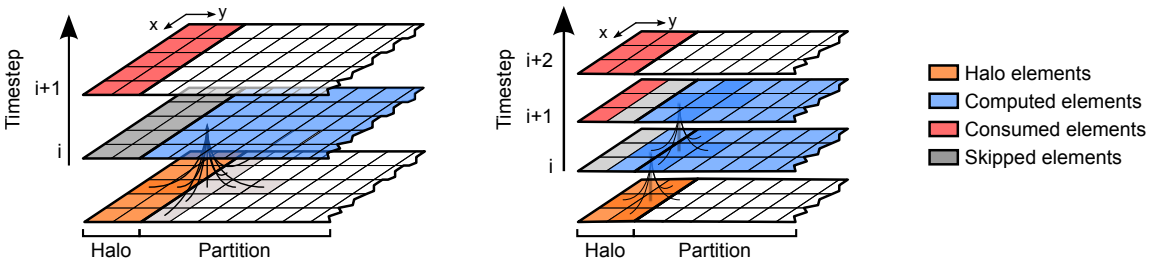


Figure 2.16: Halo consumption in a 2D domain. When decomposing a stencil, the tiles have to overlap to satisfy the spatial dependencies at the edges. The halo has to be at least as large as the stencil shape (size 2 on the left). Oversized halo allows for multiple timesteps to be computed before the halo is consumed (right).

a larger stencil share – shown on the left-hand side – consumes the entire halo at once, so only a single timestep can be computed.

Once a halo region has been consumed, it must be updated with the fresh values from the neighboring sub-domains. This process is called *halo swapping*. It requires a synchronization and data exchange between the devices, which impairs performance and limit scalability. To reduce the frequency of these swaps, the size of the halo can be increased, which allows multiple timesteps to be independently computed.

However, increasing the amount of overlap has two negative consequences. First, the elements in the halo regions are effectively processed multiple times by different tiles, which introduces memory access and computation redundancies. Second, the cost of the communication themselves is greater, since the size of the halos to swap has increased.

Therefore, there is a trade-off between the benefits from processing sub-domains in parallel and the overheads it introduces.

2.7 SUMMARY

This chapter has provided background information about the main architectures encountered in heterogeneous systems, and the programming models used to abstract them. In particular, the OpenCL model has been described in detail and its limitations exposed. More focused background work was presented to define the terminology and techniques used in the following chapters.

The next chapter describes research related to improving programmability of heterogeneous systems.

3 RELATED WORK

This chapter introduces the state-of-the-art research which influenced our work. There is a significant body of related work in the field of heterogeneous computing. However this chapter focuses mainly on GPU programmability and its evolution since the development of the programming language Brook [Buc+04] in 2004. The use of GPGPU programing has become increasingly popular since then, and a wide variety of tools and techniques have been developed.

Section 3.1 presents the prior work investigating generic solutions to the issues of programmability and performance portability for heterogeneous systems, and on GPUs in particular. The section covers a range of optimization techniques and abstraction frameworks; illustrating the impact of the required transformations on the user code and how this can be mitigated by libraries.

Section 3.2 introduces the work related to Helium – the dynamic OpenCL optimizer presented in Chapter 4. The section covers dataflow analysis using the OpenCL model and compiler-based optimizations techniques applied on classical and heterogeneous systems. Some influential insights about JIT compilation and staging are also discussed in the same section.

In Section 3.3, we discuss research relevant to stencil computations and the Partans framework. The section presents general optimization techniques for this particular pattern on various architectures. We present existing efforts towards improving performance on single and multiple devices. The main stencil computation abstraction frameworks, compilers and code generators are then presented and compared.

Sections 3.2 and 3.3 each discuss the limitations of prior works and the contributions of Helium and Partans are presented in their state-of-the-art contexts.

3.1 HETEROGENEOUS SYSTEM OPTIMIZATIONS

Heterogeneous systems present a considerable programming challenge on all fronts: productivity, performance and portability. The knowledge of complex architecture, level of expertise required and abstraction necessary to reconcile often contradictory optimization strategies have been hinted in Section 2.2.3. This section presents recent research providing general insights and techniques used to remedy this situation.

Section 3.1.1 lists abstraction frameworks and non intrusive techniques improving productivity or providing a simplified and unified interface to software developers. Section 3.1.2 presents prior work on GPGPU optimization techniques. Both provide a complementary approach to improve programmability for heterogeneous systems.

3.1.1 *Abstraction Frameworks*

Low-level optimizations can be hidden from the users, either by proposing high level APIs, or by providing transparent software stacks. These approaches decouple the core semantics of the program from the optimization process, improving productivity and achieving reliable performance. This section discusses both techniques.

HIGH LEVEL LIBRARIES Many libraries try to hide the complexity of heterogeneous systems by providing a very high level interface to the user. These are usually based on algorithmic skeletons, which provide convenient semantic building blocks without being tied to a specific implementation, giving the underlying framework a high degree of freedom in the optimization and tuning space. **SkePU** [EK10], **SkelCL** [SKG11] and **Muesli** [EK12] are three similar skeleton based libraries where users can combine algorithmic patterns such as *Map*, *Zip*, *Reduce* or *Scan* to create more complex applications.

The three libraries adopt comparable abstraction concepts. Each pattern requires an element function, which is a unary or binary operator. It is specified as a plain text string in **SkelCL**, defined using a macro based DSL in **SkePU** or written directly in CUDA for **Muesli**. The libraries then combine specific high level knowledge about the skeletons and the element functions to generate optimized code. **Muesli** and **SkePU** can generate OpenMP and CUDA code; **Muesli** additionally provides MPI support, which allows the distribution of the computation on GPU clusters. **SkelCL** is based solely on OpenCL.

Data consistency and communication between the host and the device is achieved automatically in **SkePU** and **SkelCL** using special parallel container types, and the process is partially automated in **Muesli** using user-defined container types and explicit communication strategies. Domain-specific knowledge about the skeletons allows the three libraries to split input arrays safely and distribute the computation amongst multiple devices with a minimal impact on the application source code despite the underlying complexity.

OPENCL IMPLEMENTATIONS Some research efforts aim to extend the functionality of existing OpenCL applications automatically, without modifying the source code. These approaches provide alternative OpenCL implementations, seamlessly substituting the vendor implementations. The customized runtimes exploit some characteristics of the OpenCL model to transparently distribute computation.

SnuCL [Lee+10; Kim+11; Lee+11] automatically distributes computation written for a single device to all available devices on the same host or even on a cluster. It also performs load balancing amongst the devices by profiling a sample run. The framework extracts semantic information from the OpenCL features. A source-to-source translator then generates an optimized program in a different paradigm – in this case C and CUDA.

FluidiCL [PG14] also provides an OpenCL runtime to distribute single-device application between a CPU and a GPU. Optimizations and compiler transformations automatically improve performance and communication between the host and the device. The main limitation of **FluidiCL** is that kernels cannot execute asynchronously.

dOpenCL [KSG13] exposes all devices in a cluster as a single unified host platform. Multi-device OpenCL application scale seamlessly to multiple hosts, without the burden of writing complex code to handle the multiple communication layers. The framework automatically generates MPI communication between the hosts to emulate the OpenCL memory operations as if it were happening on a single host.

These high-level abstraction frameworks leverage a collection of optimizations, and present a common interface to hide their complexity. As such, this approach is easily extended by integrating novel techniques or targeting new devices. The applications built on top of the abstraction are oblivious to these changes, and enjoy better performance transparently as the underlying frameworks are becoming more advanced. The next section describes the most common optimization techniques used for GPU programming.

3.1.2 *GPGPU Optimization Techniques & Insights*

New optimization techniques have to be developed for each architecture in order to achieve peak performance. This is especially the case for GPUs, where the architecture is radically different from traditional CPU hardware. This work is also motivated by observations that the performance gap between optimized and unoptimized code is several orders of magnitude. The main optimization techniques are described below.

MEMORY COALESCING Optimizing memory access patterns is of primordial importance on most GPU architectures. The architectural reasons for this requirement and the concept of *memory coalescing* have been introduced in Section 2.2.2. A lot of research explored ways to guarantee memory coalescing, either enabling it by construction through abstraction or introducing it in non-coalesced code via compiler transformations.

Sung, Stratton, and Hwu [SSH10] explored different multidimensional volume flattening strategies for structured grids. A code analysis pass extracts the memory accesses for each array in a kernel function and expresses them in an affine form. If the access patterns match some predetermined criterion, an optimized flattening function is generated by combining rotation and padding transformations. The resulting layout is compared to structure-of-array (SoA) and array-of structure (AoS) layouts, achieving a speedup of up to 30% over the best of the two in their evaluation. For matrices, the generated code also outperforms both row-major and column-major representations by a slight margin.

The same authors performed a careful analysis of SoA versus AoS layouts in terms of memory bandwidth [SLH12]. The authors propose an alternative layout strategy called Array-of- Structure-of-Tiled-Array (ASTA), which preserves memory coalescing, like SoA, while limiting the distance between multiple fields in complex datastructures, which is the strength of AoS. This is achieved by slicing an AoS representation and using a SoA layout within the tiles. The transformation is done on-the-fly at runtime while the data is transparently marshaled to be sent to the GPU, at a much lower overhead than AoS to SoA transformation. The proposed layout achieved roughly the same performance as SoA on the device but is substantially faster when factoring in the communication costs.

Jang et al. [Jan+11] investigated similar data layout optimizations to improve memory coalescing for loop nests. They implemented rotation, shift and strided access as generic compiler transformation rules. The optimizer is free to generate multiple versions of the

same array and apply a distinct optimization set to each copy in order to improve different accesses. In addition to layout transformation, global memory can be promoted to a more specialized space like texture or constant memories whenever possible to further increase performance. In the same paper, the authors show that vectorization is also a very effective transformation to improve memory usage by avoiding thread stalls. The two transformations target different types of accelerators: improving data layout is more efficient for scalar-based architectures like Nvidia GPUs, while vectorization is more effective on vector-based architectures like AMD GPUs.

THREAD COARSENING GPU kernels routinely launch hundreds of thousands of threads. This is necessary to exploit a high degree of parallelism. However, it might be inefficient in terms of resource utilization. Some kernels exhibit a large amount of redundancies across threads. It is the case, for example, of common memory requests, or a high control-flow to floating-point ratio in the case of fine grained computation. An optimization known as *thread coarsening* [USQ12] consists of evaluating multiple work items within a single thread.

AUTO-TUNING The optimal set of parameters is hard to find for any problem. However, they are necessary to maximize device occupancy. Many factors impact performance, and they are not necessarily intuitive even for expert programmers. Automated techniques are an efficient solution to avoid manually navigating in a complex space.

Auto-tuning can be used to adapt the OpenCL model automatically to a given architecture. For example, Magni et al. [MDO13; MDO14] developed a thread coarsening compiler transformation coupled with a machine learning based auto-tuning mechanism to choose the best thread coarsening factor. This approach leads to an average performance improvement of 33% for a wide range of benchmarks across different device types.

The same techniques can be applied to algorithm parameters, which can be explicitly provided by the user or automatically discovered through profiling.

GADAPT [LZS09] is an auto-tuning framework for CUDA applications. It predicts near-optimal configuration for a range of pragma-annotated parameters. A model is trained to recognize performance critical patterns in the code and their tuning parameters as well as their sensitivity to compiler transformations such as tiling and loop unrolling for a range of inputs. The result of the optimizer is an input-adaptive program which uses a performance predictor to select optimal parameters for a yet-unseen input.

Auto-tuners can also be integrated to high level abstraction libraries. This provides a non-intrusive way to improve performance for applications using these APIs.

Seo et al. [Seo+13] investigated a profiling-based tuner integrated to **SnuCl** to select an optimal workgroup size on multicore CPUs. Choosing efficient index space decompositions allows an improved data locality and enhances the effect of other transformations like thread coarsening. Memory accesses gathered during a profiling stage are represented in a polyhedral model. This information is then used to assess cache usage and load balancing, which are optimized for a particular architecture. A code generator can produce different variants of the code with distinctive characteristics. A heuristic-based algorithm simplifies the search space, and the remaining versions of the code are exhaustively explored in order to find the optimal optimization set.

Dastgeer, Enmyren, and Kessler [DEK11] investigated an auto-tuning framework for **SkePU** using machine learning and performance prediction. Communication costs and kernel runtimes are estimated using micro-benchmarks. An off-line tuner then predicts the optimal domain decomposition on GPUs or the optimum OpenMP parameters accurately.

DISTRIBUTED COMPUTATION Using multiple devices in a single application is advantageous when the computation can be divided into independent tasks. However, scheduling strategies must be implemented to choose the devices, determine an efficient mapping and dispatch computation.

Augonnet et al. [Aug+10] integrated a data-aware scheduler to the **StarPU** framework. The authors developed a data transfer overhead predictor which impacts the scheduling decisions: since data transfer to and from a dedicated accelerator might overshadow the performance benefits, the ability of assessing these costs in a complex task graph can dramatically improve performance. The task scheduler also exploits the information from a task graph to overlap a task execution with memory operations required by a subsequent task, overlapping computation and communication.

Alexandre, Marques, and Paulino [AMP14] developed a dynamic scheduler supporting multiple GPUs based on the skeleton library **Marrow** [Mar+13]. They rely on micro-benchmarking to estimate the relative performance of each device in a heterogeneous system in order to guide the work distribution and domain decomposition. The authors report that the PCIe bus, used as the de-facto interconnect medium between multiple dedicated accelerators is the main scalability bottleneck for distributed computing.

3.2 COMPILER AND DYNAMIC OPTIMIZATIONS

The previous section presented generic optimization techniques and frameworks for heterogeneous systems. These mechanism show promising results. However, they are highly intrusive and not easy to put in place. Other approaches try to integrate these optimizations into existing code via compilers or automated translation. The Helium framework, presented in Chapter 4, is an example of a dynamic OpenCL optimizer which applies optimizations in a non-intrusive way by building a task graph at runtime and using a JIT compiler to generate staged optimized device code. This section presents the most relevant work in the areas of JIT compilation, dynamic optimizations and code staging.

3.2.1 *Compiler Analysis & Profilers*

Languages like OpenCL and CUDA lack a unified view of the entire execution flow. The device and host codes are compiled and executed in isolation, creating an information gap. Some frameworks address this by performing a system-wide program analysis.

Mistry et al. [Mis+11] developed a profiling technique for analyzing data flow in multi-kernel OpenCL applications. The output is a timeline where the start and end time of each kernel invocation is plotted, as well as the time it was pushed in the queue. Their framework does not automatically improve performance but allows programmers to identify bottlenecks by manually inspecting a profiling trace. One of the examples they provide highlighted the different behaviors of a single application on various architectures. The exact same sequence of OpenCL commands has different bottleneck depending on the device.

Jablin et al. [Jab+11] explored a CPU-GPU communication framework to track buffer usage and manage memory allocation and transfers between host and devices automatically for CUDA programs. Explicit communication is removed from the applications and their framework infers the communication patterns from the kernel invocations and optimize them. This scheme unwinds complex cyclic communication patterns into acyclic schemes, improving data locality and decreasing the number of memory transfers between host and device.

3.2.2 *Compilers for Heterogeneous Systems*

Some research projects investigated the possibility to use source-to-source compilers to adapt existing parallel paradigms to heterogeneous systems automatically. This is a very difficult task since all paradigms are slightly different and exploit different aspects of parallelism. A common approach to this problem is to extract a high level representation of the code to transform it without modifying the semantics of the application.

Lee, Min, and Eigenmann [LME09] described a compiler framework for automatic source-to-source translation of OpenMP code to CUDA. It uses model similarities between loop level parallelism from OpenMP and the CUDA grid parallelism. The translation is divided into two phases: the first stage optimizes the OpenMP annotations and transforms the blocks into equivalent constructs which are friendlier to common GPU architectures. The second phase translates the annotated blocks into CUDA by generating kernel functions.

Grewe, Wang, and O'Boyle [GWO13] explored a similar approach for OpenMP to OpenCL transformations. Their compiler translates the annotated regions of code to OpenCL and integrates it to the OpenMP implementation. A runtime system coupled to a performance predictor makes the decision to run either the OpenMP or the OpenCL code at runtime, depending on the runtime context of the application. They also investigated automatic compiler optimizations of translated OpenMP code to match the GPU architecture more efficiently. The transformations proposed include loop rotation, memory indexing optimizations and local memory prefetching.

3.2.3 *Dynamic Optimization & Staging*

Optimizing code at runtime leads to better performance than any statically optimized code. The transformations take advantage of runtime values and invariants which cannot be exploited statically. This process is called dynamic optimization and relies on dynamic compilation, which can be divided into two categories. The first category is just-in-time compilation, where the compilation from source code is performed ahead of the initial execution but delayed until then. The second is selective dynamic transformation. It is a form of staged compilation where only a subset of code – usually delimited by user annotations – is iteratively compiled. The work presented in Chapter 4 is based on techniques borrowed from both approaches, for which the related work is described below.

JIT COMPILATION Compilation can be delayed until the code has to be executed. It can then use the execution context which was not accessible statically for optimizations.

OpenCL and the forthcoming CUDA 7 [Har15] embed a just-in-time compiler to generate device code at runtime, improving portability across devices. Just-in-time compilers have been used in other contexts and languages and have been extensively studied.

Adl-Tabatabai et al. [AT+98] and Krall and Grafl [KG97] independently implemented JIT compilers integrated into the Java Virtual Machine (JVM). They both generate native code from Java Bytecode after a single linear traversal of the instructions. They implement the same optimizations: efficient register allocation, common subexpression elimination and static predicate evaluation.

Dubach et al. [Dub+12] proposed **Lime**, an extension to the Java array notation allowing a JVM compiler to offload computation to an accelerator. It uses properties of the Java language such as immutability and strong isolation in order to prove the absence of side effect in a code fragment, hence the legitimacy of decoupling the computation. The annotated source code is analyzed and used by a static compiler to generate OpenCL code and the Java code required for interoperability. Data layout transformations and communication with the accelerator are fully automated and optimized.

SELECTIVE DYNAMIC TRANSFORMATION & STAGING Re-compiling the same region of code in specific contexts allows a different set of optimizations to be applied. Switching between several specialized version of a code fragment typically outperforms any static or JIT optimizations. This technique is generally used to accelerate languages where compute intensive parts of an application can be isolated or marked for optimization.

Recent JVMs use complex dynamic optimization frameworks in order to identify frequently executed sections of Java Bytecode, or *"hot spots"* and specialize them or generate native code. **Jalapeño** [Arn+00] and **Graal** [Gra] are the most recent compiler extensions for the JVM. **Jalapeño** performs relatively safe transformations using flow insensitive optimizations while **Graal** applies more aggressive speculative optimizations and must implement a cautious fallback mechanism in case the specialization is not applicable.

Project Lancet [Rom+13] is a framework for interacting with the Java JIT compiler though a high level API. Instead of automatically inferring dependencies and optimizations, the Lancet compiler relies on user annotations and explicit specialization, allowing users to

control the transformations. For example, variables can be *"frozen"* to indicate that they become invariant and are strong candidates for optimization.

Beckmann et al. [Bec+04] presented a multi-stage programming framework for C++. Users invoke a JIT compiler explicitly to partially specialize specific regions of code. This enables compiler optimizations such as constant propagation, loop unrolling or vectorization to make use of runtime values. It also allows users to define task graphs where the actions are collectively optimized, generating cross-action optimizations like loop fusion.

3.2.4 *Limitations of Prior Work and Discussion*

The works described in this section tend towards a common goal, which it to increase the applicability of compiler transformations, either offline or at runtime. Each technique has distinct strengths and weaknesses, which are discussed in this section. We also introduce some of the contributions made by Helium, the dynamic OpenCL optimizer described in Chapter 4. in order to position it in the vast landscape of existing tools and techniques and distinguish it from prior work.

While source-to-source translators are attractive since they don't require any code base change, they have limited applicability and success. This is due to the difficulty in efficiently translating concepts from one paradigm to another. Furthermore, they are heavy transformations which can be error prone and slow. Thus, source-to-source compilers are not well suited to improve adaptability of dynamic or complex applications. However, since they have to extract high level semantic information from the input code in order to translate it, they can perform very aggressive optimizations and transformations, like restructuring large parts of the code. These optimizations are rarely achieved using other techniques since they require a global analysis and a high level understanding of the code which is not present and very difficult to recover from a lower level.

JIT compilation is a much more efficient technique to improve adaptability of the code, as demonstrated by OpenCL and CUDA. Delaying computation can take advantage of runtime knowledge such as runtime code invariants or even the type of device targeted to guide compiler optimizations in a way that outperforms any static optimization. However, the high overhead introduced by integrating the full compilation pipeline makes it impractical for staging. This limits the applicability of one-off ahead-of-time compilation

for highly dynamic applications where specializing the code at different points of the execution would result in much more efficient optimization sets due to the specific contexts.

Selective dynamic optimizations and staging provide an answer to this problem: they are strategies complementary to ahead of time compilation (either static or runtime), where the same transformations are applied multiple times, creating specialized variants of the code. By operating at a low level, they are typically more lightweight than JIT compilation. However, these techniques often require users to hint the optimizations, either by instrumenting the code or by taking control of the JIT compiler, which severely limits applicability and efficiency. Optimization candidates can be discovered automatically using trace profiling, but it creates a high overhead phase. The other major drawback of profile-driven staging is that the code is often speculatively optimized, which entails two limitations: specializing a region of code is only beneficial if it is executed again after the optimization and a number of safeguards have to be in place to ensure a safe fallback in case the optimization does not fully match the planned execution, which introduces overheads.

In contrast to the existing approaches, we combine techniques from all three levels to develop a dynamic OpenCL optimizer, Helium.

Similarly to a source-to-source compiler, Helium reconstructs a very high level representation of the program semantics to guide the optimization process and guarantee the validity of aggressive transformations such as kernel fusion. Instead of implementing complex and fragile static analysis to recover the program semantics, Helium uses the discrete-memory and kernel-based OpenCL model to build the task and dataflow graphs during the program execution. This information can be taken into account during the runtime compilation phase.

OpenCL device code can be optimized against runtime information using the JIT compilation. However, Helium pushes this model further by keeping a low-level representation of the device program in its embedded compiler.

This intermediate representation is used throughout the application to implement lightweight staging. Each execution site is further optimized, using the runtime knowledge available then. Performing the transformations at a low-level presents several advantages. First, it introduces negligible overheads compared to full-blown compilation – which is the main strength of dynamic optimizations. Second, high-level runtime information is used to guide the optimization process and avoid profiling and speculation overheads.

By packaging the optimizer as an OpenCL runtime, it can be readily deployed over existing OpenCL binaries to significantly and transparently improve performance, in the same way **SnuCL**, **FluidiCL** and **dOpenCL** can easily be deployed.

Abstraction frameworks like **SkePU** and **SkelCL** enable very efficient optimizations and are a good solution for boxing underlying heterogeneity and hiding implementation details. Most of them are built on top of core parallel paradigms like OpenCL and CUDA instead of defining their own model, which makes them more versatile. Hence any transparent optimization targeting the underlying mechanism is orthogonal to the high level optimizations. This makes Helium not only beneficial for application using directly the OpenCL API but also complementary to any OpenCL abstraction library.

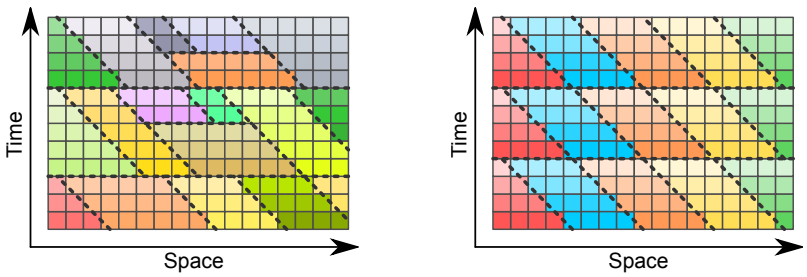
Helium is the first dynamic optimizer to our knowledge to combine high-level information – like task graph representation – and low-level techniques – like iterative compilation and staging – to optimize a widely used heterogeneous parallel programming paradigm like OpenCL transparently. Its implementation and evaluation are described in Chapter 4.

3.3 STENCIL COMPUTATION

Stencil computation is a fundamental compute-pattern in many domains and has been extensively studied. Section 2.6 presented some of its characteristics and challenges. This section presents some of the existing optimization techniques for heterogeneous systems. State-of-the-art tools and abstraction layers improving programmability of stencil code are introduced.

Since Partans – our high-performance stencil computation framework presented in Chapter 5 – targets primarily multi-GPU systems, this section focuses more on distributed stencil optimizations rather than single-device convolution optimizations.

A comprehensive survey of existing abstraction frameworks for stencil computation is provided in order to compare their features and interface. Their strengths and limitations are discussed and the contributions of Partans are put into this context.



(a) Cache Oblivious Tiling (b) Time Skewing

Figure 3.1: Cache Oblivious(a) and Time Skewing(b) optimizations: the 1D-time 1D-space spacetime domain can be recursively decomposed by cutting either along the space or time dimensions. This transformation is always correct as long as dependencies between tiles (left to right and top to bottom) and temporal dependencies (dark to light) are respected.

3.3.1 *Stencil Optimization Techniques*

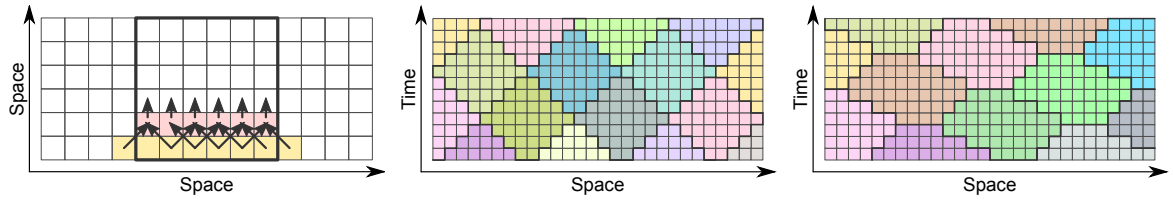
This section focuses on background research on general optimization techniques for various architectures and distributing stencil computation across multiple devices.

The common goal of these techniques is to maximize data locality. The generally low computation to memory access ratio makes most stencil problems memory-bound. While convolution is a very simple compute pattern in its naive form, most optimizations require substantial transformations.

TILING In its most naïve form, stencil computations are essentially an element function applied within a loop nest. Hence it can be optimized using general-purpose loop compiler transformations like rotation [MCT96], skewing [WL91], fusion [KM94; Dar99] or tiling [IT88; GAK03]. These techniques by themselves are shown to enhance performance considerably by improving data locality.

Advanced iteration-space decomposition schemes can take advantage of domain-specific knowledge to further improve performance. Tiling in both the spatial and temporal dimensions opens up additional opportunities for cache reuse across iterations.

Cache oblivious methods [FS05] increase cache reuse in an architecture-independent way by recursively cutting the spacetime domain. The domain can be split along the time dimension and space dimensions. Note that the space cut needs to be skewed in the time dimension as well to satisfy the dependencies of the boundary elements. These two cuts can be applied recursively until a tile fits into a cache or the minimum space and time dimensions dictated by the stencil operator are met. Figure 3.1a illustrates a possible de-



(a) 2.5D cut in 2D space (b) Split tiling in 2D spacetime (c) Hexagonal cut in 2D spacetime

Figure 3.2: Tiling methods maximizing parallelism: these spacetime cuts aim to increase the degree of parallelism while maintaining good data locality. While there are still some spatio-temporal dependencies between the tiles, all tiles at the same level in the spacetime domain are independent and all the elements within a tile can be processed in parallel at a given timestep.

composition resulting from applying multiple cuts. Horizontal slices are time cuts while vertical divisions represent cuts in the space dimension. While the time cut does not improve locality directly, it allows the space cut to be applied a greater number of times in cases where the tile spans many elements in time but not in space.

Time skewing [Wonoo], is a similar approach but the cuts are precisely controlled. Identical space cuts are applied with a stride. This creates regular tiles, except at the edges. Like the space cut in the cache oblivious optimization, the first and last ones are irregular due to time skewing. However, all the tiles in between are typically identical when applying controlled time skewing. This approach allows multiple iterations to be computed by a block. In order to satisfy dependencies at the edges of the tiles, the number of iterations is limited to the height of the first. A time cut then resets the cycle. A time skewing decomposition is shown in Figure 3.1b

The *Circular Queue* optimizations [Yan+12] consist of prefetching as many slices of a volume tile as necessary to satisfy the stencil shape and storing them in a sliding window structure. This allows computation of one slice of the output volume. The first input slice is then discarded from the local buffer, but only the next slice needs to be fetched to compute the following output slice. The rest of the buffer is re-used across iterations. This method introduces task parallelism since there is no space dependency between the tiles, unlike time skewing. However, the elements at the tile boundaries are processed by multiple tiles, introducing redundant computations.

The tiling techniques described thus far are primarily intended to enhance data locality in a restricted level of parallelism, which suits CPU architectures well. On GPUs however,

the level of parallelism is tremendously higher and other tiling methods taking advantage of that fact were developed. These strategies are presented in Figure 3.2.

2.5D and 3.5D tilings [Ngu+10] combine spatial decomposition into independent tiles which can be computed in parallel and a wavefront pattern within each tile to enable reuse of fetched elements. Adding several wavefronts and increasing the amount of overlap between tiles enables the computation of multiple timesteps using time tiling with an implicit time cut between each iteration. However, this increases the amount of redundant computation at the tile boundaries. Figure 3.2a shows an example of 2D space cut along one dimension with a 1D wavefront across two space iteration: this is 1.5D space, 1D time decomposition – called a 2.5D blocking scheme using the author's terminology.

Hyperplane cuts [BPB12; Gro+13] or hexagonal cuts [Gro+14] allow all tiles and timesteps to be independent for a limited number of iterations. All elements of the domain are computed in parallel, but it requires careful synchronization at the tile boundaries. Maintaining coherent data is achieved by enforcing an ordering in the tile computation. In Figures 3.2b and 3.2c, they should be processed from the bottom to the top. Alternatively, a time cut is inserted between each time step to synchronize the tiles. Some hybrid system combine both space and time cut to create complex tiling patterns.

Tiling methods can also be adapted to take advantage of the underlying architecture. Meng and Skadron [MS11] explore a warp-aware tiling implementation for Nvidia GPUs. Knowledge about the device and the stencil properties can be used to predict efficient decompositions which maximize resource utilization. While this technique nearly reaches the theoretical peak performance, it is very hardware dependent and creates code for which not only the performance but also the correctness is tied to a particular device.

DISTRIBUTED STENCIL COMPUTATION The presence of spatial and temporal dependencies makes distributed stencil computation an engineering challenge. This complex dependency pattern requires careful synchronization and communication between the participating actors. Domains can be decomposed using essentially the same single-device tiling techniques described above, but this introduces hierarchical communication, which easily become a bottleneck.

Micikevicius [Mic09] describes a distributed stencil computation on multi-GPU systems to solve the 3D finite difference problem. Their system uses MPI to communicate between

multiple hosts. Each device prioritizes computation at the edges of the tiles and initiates communication with their neighbor while computing the inner part of the tiles in order to overlap computation and communication. The authors report superlinear speedups for two and four GPUs due to a decreased pressure on the Translation Lookaside Buffer (TLB) buffers when the data is distributed. They also claim to entirely amortize the communication overheads between the GPUs after manually tuning the global and local tile sizes and the amount of overlap between tiles.

Zhang and Mueller [ZM12] is another GPU cluster implementation based on MPI. The authors observe that communication pressure is the bottleneck and is highly sensitive to stencil characteristics, in particular the order of the stencil shape which directly affects the computation to communication ratio. They observe quasi-linear speedup for a first order 27 point 3D stencil on up to 15 GPUs. The performance degrades significantly for a higher number of GPUs, a higher order or a smaller shape.

Göddeke et al. [Göd+08] also uses a MPI-based communication layer to extent parallelism to multiple hosts. The authors report the PCIe bus as the primary bottleneck for single-host systems and an impediment to scalability, in the same way network communication across hosts is a limiting factor for stencil computation distributed on clusters.

AUTO-TUNING The tuning space of any stencil computation is enormous since many optimization strategies described so far have several parameters each. In particular, choosing a data layout or the tiling strategy and size for distributed and local decomposition is not trivial. Auto-tuners can automatically navigate and in some cases prune this space in order to determine optimal values.

Datta et al. [Dat+08] performed a very thorough evaluation of a multitude of optimizations on various CPU and GPU architectures. The authors discuss the strengths and weaknesses for each device depending on the stencil characteristics. Their experiments show that stencil computations exhibiting enough parallelism benefit more from a large number of simpler processors, rather than a small number of more complex cores. The study also shows that auto-tuners are critically important to reach good performance for stencil computation. Manual performance improvement is very difficult since most of the space is largely and uniformly sub-optimal, leading programmers to believe that they reached peak performance on the targeted device prematurely.

Framework	Programming Language		Parallelism Scale			
	Frontend	Backend	Cluster	CPU	GPU	Multi-GPU*
Cactus [RIF01]	DSL	C/CUDA	✓	✓	✓	×
STELLA[Fuh+14]	Fortran	CUDA	✓	✓	✓	✓
Forma [RHG15]	DSL	C++/CUDA	×	✓	✓	×
OverTile [HPS12]	DSL	CUDA/OpenCL	×	✓	✓	×
PADS [Han+11]	OpenMP	CUDA	×	×	✓	×
PATUS [MC11]	DSL	OMP/CUDA	×	✓	✓	×
Physis [Mar+11]	DSL	CUDA	✓	×	✓	×
Pochoir [Tan+11]	C++	Cilk	×	✓	×	×
SBLOCK [BP10]	DSL	C/CUDA	✓	✓	✓	×

* *Multi-GPU support within a single process (i.e. not distributed).*

Table 3.1: Comparison of frameworks for stencil computation on heterogeneous systems. For each framework, we report the input language used for the stencil specifications and the code generated by each compiler. The target for each framework is also specified.

Zhang and Mueller [ZM12] implemented an auto-tuner to exhaustively search the decomposition space for single device tiling. This generates profiling estimation which is used by a multi-node tuner to balance communication and computation efficiently based on the performance of each individual device. Their optimizer hides communication latency, allowing them to scale to multiple GPUs almost linearly until the communication overheads overtake the computation time.

3.3.2 *Frameworks, Code Generators & Compilers*

Abstraction layers have been successfully designed to decouple optimizations from the essential semantic specification of a program. Examples of general-purpose skeleton-based libraries for heterogeneous systems have been described earlier. This section focuses on similar frameworks and tools for stencil computation. Most of these abstractions use resembling interfaces. The user only defines the essential properties of the stencil, such as the element function and the domain specification. The optimizations are then automated using code generators or compilers. The main frameworks for stencil computation on heterogeneous systems are listed in Table 4.1. Their input languages are compared, as well as the underlying paradigms used to express the optimized code. They target different scale of parallelism, from multi-core CPUs to cluster of accelerators. This section briefly describes each of them and hints some details of their implementation.

```
stencil heat3d
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;
  operation(float grid u,float param a,float param b)
  {
    u[x, y, z; t+1] = a*u[x,y,z;t] + b*(
        u[x-1,y,z;t] + u[x+1,y,z;t] +
        u[x,y-1,z;t] + u[x,y+1,z;t] +
        u[x,y,z-1;t] + u[x,y,z+1;t] );
  }
}
```

(a) Patus [MC11]

```
Pochoir<float,3,1> fd_3D ;
Pochoir_Array<float,3,1> u (Nx,Ny,Nz);
Pochoir_Shape<3> fd_shape_3D [] = {
 {1,0,0,0},{0,0,0,0},
 {0,-1,0,0},{0,+1,0,0},{0,0,-1,0},
 {0,0,+1,0},{0,0,0,-1},{0,0,0,+1}};

Pochoir_Kernel_3D( lap,t,x,y,z )
  u(t+1,x,y,z) = a*u(t,x,y,z) + b*(
      u(t,x-1,y,z) + u(t,x+1,y,z) +
      u(t,x,y-1,z) + u(t,x,y+1,z) +
      u(t,x,y,z-1) + u(t,x,y,z+1) );
Pochoir_Kernel_End
```

(b) Pochoir [Tan+11]

```
input scalars = ["a", "b"]
input arrays = ["u"]
output arrays = ["v"]
inner loop = [{
 "lvalue": "v",
 "rvalue": a*u[0][0][0] + b*(
   u[-1][0][0] + u[1][0][0] +
   u[0][-1][0] + u[0][1][0] +
   u[0][0][-1] + u[0][0][1])
}]
```

(c) SBLOCK [BP10]

```
program heat3d is
grid 3
field u float inout

u[1:1][1:1][1:1] =
 0.12*Eu[0][0][0] + 0.45*(
   u[-1][0][0] + u[1][0][0] +
   u[0][-1][0] + u[0][1][0] +
   u[0][0][-1] + u[0][0][1])
```

(d) OverTile [HPS12]

```
stencil head3d(float a, float b, vector#3 float u) {
 return a*u + b*(u@[-1,0,0] + u@[1,0,0] + u@[0,-1,0] +
   u@[0,1,0] + u@[0,0,-1] + u@[0,0,1]);
}
parameter a,b,Nx,nY,nZ;
vector#3 float u[Nx,Ny,nZ];
v = head3d(a,b,u:constant(0));
return v;
```

(e) Forma [RHG15]

Figure 3.3: Example of 3D heat equation element function written in existing frameworks for stencil computation on heterogeneous systems. Most frameworks decouple the stencil operator and the specifications for the domain and the iterations (not shown).

Christen et al. developed a domain-specific language named Parallel AutoTUned Stencils (**PATUS**) [CSB11; MC11]. A code generator coupled with an auto-tuning infrastructure enables the translation of high level stencil specifications into highly optimized architecture specific code. Optimizations such as tiling and vectorization are applied automatically, and the generated parallel code can be executed on multi-core CPUs or GPUs using either an OpenMP or a CUDA backend respectively. An auto-tuning mechanism automatically explores a parameter space determined by an execution strategy to optimize its parameters. Users-defined strategies can be implemented to guide the auto-tuning phase.

Pochoir [Tan+11] is a framework and a compiler for stencil computation. It provides a C++ API which allows code to be compiled and executed by any compiler. The same code can be translated by their compiler to a highly optimized Cilk version, automatically introducing tiling and cache-oblivious optimizations.

Holewinski developed a similar approach for GPUs. A compiler translates high-level specifications, written in a special stencil language called **OverTile** [HPS12], into optimized CUDA code.

Halide [RK+12; RK+13] is a domain specific language, a compiler and a runtime system targeting mainly image processing applications. Stencil operators are defined using their DSL as functions over an infinite, unbounded domain; this forms an independent intrinsic algorithm. Separately, schedules can be defined to map algorithms to materialized domains and create pipelines. The optimization process is semi-automatic: the user defined schedules define partial transformations and their compiler generates the required low level code. Because schedules are expressed at a very high level and independent from the algorithm specification, changes that would require a near complete re-write of a C implementation can be contained to a couple of lines in the schedule specification.

SBLOCK [BP10] is a stencil-based PDE solver library. It targets CPUs, clusters of multi-cores and GPUs. The library is divided into two parts: the run-time library and the source code generator. The run time library is in charge of inter-node communication using MPI and more fine grained scheduling. It provides an interface in C for the user to define the volume description and highlight an optimization strategy. The source code generator translates at the user-defined kernels written in Python to an architecture specific optimized code which will be further compiled. The framework has built-in support for reduction operations and complex volume definition.

Cactus [RIF01] is a divide and conquer framework supporting different classes of scientific tightly coupled applications, including stencil computations. It relies on grid computing to do massively parallel jobs on a large scale. It is using a domain-specific language to define operators, volume description and boundary behavior for stencil computations.

PADS [Han+11] is an OpenMP to CUDA translator and a code generator using pre-implemented optimized templates. It uses a pattern-matching technique to identify stencil patterns in C++ code, which can be further enhanced by providing special annotations defined as a superset of OpenMP directives. The generated code is automatically parameterized and an auto-tuning framework explores the search space for the optimal values.

Solar-Lezama et al. [SL+07] explored automated optimizations using software synthesis. With the language and compiler SKETCH, the user provides two versions of the program: the first baseline version defines a naïve behavioral specification of the algorithm and the second outlines an optimized version with the possibility to express missing parts of the algorithms such as predicates and indexes using a '??' notation. A solver will try to implement the partially optimized version to be semantically equivalent to the baseline. By manually unrolling multiple iterations of a stencil computation, it is possible to derive optimizations such as time-skewing and cache-oblivious implementation at a fraction of the development cost and with high degree of code reuse.

Kamil et al. [Kam+10] implemented a source to source compiler allowing unoptimized stencil computation in Fortran to be automatically optimized and translated to C or CUDA. A target-specific auto-tuner optimizes the execution strategy to further enhance performance. The search space is pruned according to pre-defined heuristics to keep only the most likely optimization set for the given target and an automated optimizer exhaustively generates and executes all the candidates to find the best version.

Some libraries provide problem specific stencil-based framework. The most advanced are **Chombo** [Kam+05; VS+09], a set of tools to solve Partial Differential Equations, and the weather forecasting toolkit **COSMO** developed a high-performance stencil computation library called **Stella** [Fuh+14].

Finally, general-purpose heterogeneous programming libraries can be extended to support stencil computation. Steuwer et al. [Ste+14] investigated an extension of **SkelCL** to support distributed stencil computation. They added two patterns: a space-aware map pattern called *MapOverlap* and a specialized skeleton *Stencil* for iterative stencils. Most of the

low-level details are hidden from the user by the library's distributed container and code generator mechanisms. However, optimization parameters – like the amount of overlap between tiles – have to be set manually. **SkelPU** also provides a *MapOverlap* skeleton, but they do not provide high level support for stencils for further optimizing iterative stencils.

3.3.3 *Limitations of Prior Work and Discussion*

This section presented some of the considerable amount of research aiming to improve programmability of stencil computations, either by developing low-level optimization strategies or crafting high level interfaces to hide the implementation details and lift the programming burden from the programmers. We summarize the state-of-the-art below and introduce the design decisions and contributions of PARTANS in terms of abstraction and optimization strategies.

ABSTRACTION The most efficient optimization techniques introduced above require a substantial modification of the naïve versions. Implementing tiling strategies and time skewing methods is a long and error-prone task which leaves the original stencil specifications changed beyond recognition, leading to degraded maintainability and reusability of the code. However, these optimizations are crucial for performance and need to be carefully adapted to the targeted device.

Encapsulating stencil specifications in a domain-specific language gives more freedom to a dedicated compiler. This choice was made by the majority of the abstraction frameworks described in this section. Optimizations can be integrated into the code generation process independently, and new backends can be created to adapt to new targets. Conversely, DSLs are often over-specialized for a narrow domain. They suffer from limited applicability and are difficult to adopt due to the lack of interoperability with other languages.

The abstraction layer developed in PARTANS relies on an existing paradigm instead – in this case OpenCL. The general-purposed language is not restricted, extending the applicability of the framework. The optimizations are decoupled from the interface using data abstraction and a high-level skeleton library. This approach demonstrates that the complexity of the code can be managed. However, the main contributions of PARTANS come from the optimization strategies implemented.

OPTIMIZATION STRATEGIES Although many GPU implementations of stencil computation exist, little research has been done to scale the computation to multiple GPUs. The frameworks supporting both distributed computation via MPI and GPU implementations on single nodes could use multiple GPUs implicitly. However, having one MPI process per sub-domain impairs communication [ZM12]. Data exchanges have to pass through two levels: across nodes and then from the host application to the device[1]. **COSMO** is the only library providing an optimized multi-GPU implementation: their custom GPU aware MPI implementation recognizes GPU memory buffers and eliminates additional copies. The authors report performance benefits for MPI processes on the same physical machine. However, they did not quantify them or described the technique further. In a similar approach, Ji et al. [Ji+12] explored the impact of another GPU aware MPI implementation on halo swapping for a 2D stencil application and reported bandwidth improvement of up to 2x and an additional speedup of 5% compared to non GPU aware implementation.

PARTANS investigate this problem further by systematically exploring the tuning space generated by the presence of multiple accelerators within a single system. New tuning parameters, such as balancing the usage of communication over PCI buses as discussed in [AMP14], present an interesting challenge with potentially far-reaching benefits for the community as systems are getting increasingly heterogeneous. We develop search techniques to navigate this space and automatically optimize an application to increase the overall throughput of the system. PARTANS demonstrates how to apply these optimizations without affecting the high level abstraction. This work is described in Chapter 5.

3.4 SUMMARY

This chapter has presented prior work aiming to tame the optimization complexity and programmability issues encountered on heterogeneous systems. It detailed techniques developed to improve performance on such systems, and proposals for abstracting these optimizations in the form of high level libraries, frameworks and compilers. The limitations of existing solutions have been discussed.

The following chapter introduces the first contribution: automatically optimizing OpenCL applications using dataflow analysis, code staging and dynamic optimizations.

1 Due to lack of interoperability between the standard implementations, it is impossible to use the same pinned memory for MPI and CUDA or OpenCL host buffers, requiring additional memory copy within the host.

4 DYNAMIC INTERKERNEL OPTIMIZATIONS

This chapter presents a new approach to optimize multi-kernel applications transparently by combining static compiler analysis and dynamic runtime information. The separation of concerns between host and device code in OpenCL prevents either the host or device compiler from having a complete picture of the execution flow. This in turn prevents aggressive inter-kernel optimizations which, like interprocedural optimizations, could greatly improve data locality across kernels or minimize the use of intermediate storage. As a consequence, these optimizations are often implemented by hand despite the great effort required for refactoring the code, and at the cost of decreased kernel modularity and programmability.

This is addressed by our OpenCL overlay and runtime optimizer, called Helium. Compiler analyses of the device code and runtime profiling of the host application are combined to optimize the dataflow of the application. Using a *delay-optimize-replay* mechanism, the execution strategy of OpenCL commands is switched from eager to lazy evaluation. This allows the runtime system to apply a set of optimizations which could not be performed by hand or would negatively impact code maintainability and portability. The transformations performed are provably safe by preserving the dependencies in the data flow across operations and avoiding the introduction of data races.

Helium implements most functions of the OpenCL API and is interposed between any OpenCL application and the underlying vendor implementation, acting as a mediator and an optimizer for the command queues. Hence it can be deployed over existing OpenCL application binaries without the need for modifying or re-compiling the source code.

This chapter is structured as follows: Section 4.1 motivates this work by presenting the benefits and challenges of inter-kernel optimizations. Section 4.2 describes the optimizations opportunities in multi-kernel applications and how they are typically performed by hand. An overview of the framework is provided in Section 4.3. Its implementation is detailed in Section 4.4, where the mechanisms allowing the delay-optimize-replay strategy and the optimization algorithms are presented. The methodology used for evaluation is described in Section 4.6 and the results discussed in Section 4.7.

4.1 MOTIVATION

Section 2.4 presented the OpenCL programming model and its limitations. Performance portability was put forward as the main challenge. A great architectural diversity requires individual kernels in the device code to be manually optimized in isolation for a particular architecture. Their performance may vary when executed in a different context. However, this problem does not only affect single kernels. The combination of multiple kernels in a compute sequence creates new optimization opportunities, which are equally hard to implement in a portable and efficient way.

Inter-kernel optimizations, like interprocedural optimizations, aim to optimize a specific call sequence. This can be done by inlining multiple kernels, simplifying computation or exposing nested parallelism, which improve many aspects of the applications, in particular:

- *synchronization delays*: synchronization with the device is required in OpenCL since computation and memory operations are asynchronous. Reducing their frequency decouples host and device applications, leading to a more efficient parallelism. This can be achieved by reordering calls or executing independent tasks in parallel.
- *efficient computation*: compiler transformations like propagating constant values or merging similar control flow from different kernels decrease the amount of computation performed by the device by eliminating redundancies.
- *memory bandwidth*: identical memory accesses across multiple kernels can be factorized to decreases the overall number of transactions, increasing the useful bandwidth.
- *memory usage*: merging kernels generating data and kernels reading from it not only improves locality but also eliminates temporary storage, decreasing the memory footprint of the application.

These transformations are usually performed by a compiler in programming languages like C, where the compiler has a complete picture of the dataflow in the application; hence programmers are oblivious to these optimizations. Sadly, this does not apply to OpenCL, where the separation between device code, containing only the compute intensive fragments of an application, and the host, which orchestrate the computation without knowing how data is being manipulated by the kernels, prevents either the offline host compiler or the runtime device compiler from performing a global dataflow analysis.

However, because they might provide a significant gain in performance, these optimizations are highly desirable, and are often applied by hand. Manual tuning of complex OpenCL applications presents many challenges and is a very expensive process in terms of engineering efforts. First, identifying potential optimization candidates is very difficult or even impossible, since the programmer has to cross reference device and host code constantly to find the dataflow paths and the kernel sequence might be highly dynamic. Second, even in the case where some candidates can be found manually, applying the transformations is a very time-consuming and error-prone process. In many cases, it requires a modification of both the host and device code, which might affect the entire application. Furthermore, ad-hoc specializations of kernels to optimize specific instances create code redundancies since the generic kernel must be kept alongside the specialized variants in case the optimizations are not applicable. This considerably impairs testability and maintainability of the application and threatens code portability by running radically different versions of the program depending on runtime parameters.

In contrast, to eliminate that burden from the development process, Helium analyzes and optimizes any OpenCL application dynamically. The same transformations are applied automatically, allowing a systematic optimization process at no additional engineering cost. The original application can remain simple and modular, preserving testability and maintainability without sacrificing performance.

The next section further motivates this work by describing some interkernel transformations in more detail and demonstrating their benefits as well as the impact on the code.

4.2 DYNAMIC KERNEL SEQUENCE OPTIMIZATIONS

This section describes inter-kernel optimizations applicable on programs involving more than a single compute kernel invocation. These optimizations are often implemented manually and incrementally after the application has been developed and tested, in order to improve performance.

Throughout the section, each transformation is illustrated by a simple example. The original and optimized versions are compared side by side and differences are highlighted.

To demonstrate the performance benefit of each optimization, the provided examples are tested on an Nvidia GTX 780 GPU. Table 4.1 summarizes the speedup obtained and the main characteristics of each transformation and classes of optimizations. The required

Optimization	Required Refactoring		Potential Performance Gain			Speedup
	Host Code	Device Code	Sync. Delay	Compute Time	Memory Trans.	
Scheduling Optimizations	✔	✗	✔	✗	✗	
Parallelization	✓	×	✓	×	×	1.78x
Reordering	✓	×	✓	×	×	1.42x
Code Specialization	✗	✔	✗	✔	✔	
Constant Propagation	×	✓	×	✓	×	1.53x
Alias Resolution	×	✓	×	×	✓	1.87x
Kernel Fusion	✔	✔	✗	✔	✔	
Horizontal Fusion	✓	✓	×	✓	✓	1.71x
Vertical Fusion	✓	✓	×	✓	✓	2.11x
Task Elimination	✔	✔	✔	✔	✔	
Dead Invocation	✓	×	✓	×	×	1.72x
Dead Store	✓	✓	×	✓	✓	1.13x

Table 4.1: Summary of the dynamic optimizations. Each optimization affects the host and the device code differently and optimizes different aspects of the application.

modifications affect either the host code or the device code or both – as indicated by the same table.

Each optimization improves the overall performance. However, the potential performance gain depends on the type of transformations. As described in the previous sections, there are three main sources of overall speedup:

- *Synchronization Delay*: the compute time on the devices is unchanged but the time spent in the host application waiting for synchronization primitive is reduced. This transformation is particularly effective if the host application manages multiple devices or overlaps computation on the host and the device.

- *Compute Time*: the total amount of compute time on the devices decreases, by reducing the number of arithmetic and control-flow instructions, or improving their efficiently. Compute bound kernels would benefit most from this type of optimization.

- *Memory Transactions*: the overall number of load and store instructions decreases, either by factorizing common loads or eliminating dead stores. Reducing the number of transactions is the most efficient optimization for memory bound problems.

These optimizations are described in more detail in the remainder of this section. Code examples demonstrate the extent of the modifications for each of them. Transformations affecting only the host application are first described. Device code alterations are then presented and finally optimizations affecting both together.

```
for(int i = 0; i < 100; ++i){
  clEnqueueWriteBuffer(queue,in1,true,
    0,size,v,0,NULL,NULL);
  clEnqueueWriteBuffer(queue,in2,true,
    0,size,v,0,NULL,NULL);
  clEnqueueNDRangeKernel(queue,ker,
    1,NULL,&g,&l,0,NULL,NULL);
  clEnqueueNDRangeKernel(queue,ker,
    1,&g,&g,&l,0,NULL,NULL);
  clEnqueueReadBuffer(queue,out1,true,
    0,size,res,0,NULL,NULL);
  clEnqueueReadBuffer(queue,out2,true,
    0,size,res,0,NULL,NULL);
}
```

(a) Unoptimized Host Code

```
cl_event ec1,ec2,ew1,ew2,er1,er2;
ec1 = ec2 = er1 = er2 = NULL;
for(int i = 0; i < 100; ++i){
  clEnqueueWriteBuffer(q_write,in1,false,
    0,size,v,(ec1?1:0),&ec1,&ew1);
  clEnqueueWriteBuffer(q_write,in2,false,
    0,size,v, (ec2?1:0),&ec2,&ew2);
  cl_event dep_comp1[] = {ew1, er1};
  clEnqueueNDRangeKernel(q_comp,ker,
    1,NULL,&g,&l,(er1?2:1),dep_comp1,&ec1);
  cl_event dep_comp2[] = {ew2, er2};
  clEnqueueNDRangeKernel(q_comp,ker,
    1,&g,&g,&l,(er2?2:1),dep_comp2,&ec2);
  clEnqueueReadBuffer(q_read,out1,false,
    0,size,res,1,&ec1,&er1);
  clEnqueueReadBuffer(q_read,out2,false,
    0,size,res,1,&ec2,&er2);
}
```

(b) Hand Optimized Host Code

Figure 4.1: Hand optimization introducing task parallelism using multiple command queues and events, allowing computation and communication to happen simultaneously.

4.2.1 *Scheduling Optimizations*

Programmers can maximize parallelism using fine-grained task management. Taking control of the dependency specification between tasks, or queuing them in a very specific order enables a better cooperation between host and device programs. These optimizations exploit the task parallelism model exposed in OpenCL.

TASK PARALLELISM carefully synchronized tasks in out-of-order or multiple queues can be evaluated simultaneously. However, the efficiency of this optimization is dependent on the device since concurrency is merely suggested by the model, not enforced.

The task parallelism in OpenCL exploits *kernel concurrency* and *communication-computation overlap*. The first evaluates multiple distinct kernel instances simultaneously. The second performs independent memory operations and computation at the same time. Read, write and compute all execute in parallel on devices supporting full duplex communication.

In order to exploit this optimization, an application must manage dependencies between actions explicitly to avoid data races. Figure 4.1 demonstrates the necessary changes to introduce task parallelism as presented in the copy-compute overlap example of the Nvidia SDK. The optimized application uses multiple command queues to differentiate read, write and compute operations. Events are attached to each action to represent the dependencies. This transformation achieves a 78% speedup.

```
1 q.enqueueNDRangeKernel(CompB1,{0},{g},{l});
2 q.enqueueReadBuffer(b1,true,0,size,res_b1);
3 CrossValidate(res_b1);
4 q.enqueueNDRangeKernel(CompB2,{0},{g},{l});
5 q.enqueueReadBuffer(b2,true,0,size,res_b2);
6 CrossValidate(res_b2);
```

(a) Unoptimized Host Code

```
1 q.enqueueNDRangeKernel(CompB1,{0},{g},{l});
2 q.enqueueNDRangeKernel(CompB2,{0},{g},{l});
3 q.enqueueReadBuffer(b1,true,0,size,res_b1);
4 CrossValidate(res_b1);
5 q.enqueueReadBuffer(b2,true,0,size,res_b2);
6 CrossValidate(res_b2);
```

(b) Hand Optimized Host Code

Figure 4.2: Hand optimization reordering tasks in the command queue. Computation on the accelerator and cross validation on the host are interleaved and execute in parallel.

However, this is a very intrusive optimization which requires substantial modifications. The transformation is also delicate since an error in the dependencies will not cause a fatal error at runtime but introduce a silent data race corrupting the result. This modification also has important side effects to the rest of the application if the properties of the command queue are changed to out-of-order, and used throughout the application.

TASK REORDERING A specific execution order between independent commands is enforced; either by swapping the order in which they are pushed in an in-order queue, or adding explicit dependencies. Task reordering maximizes the potential of task parallelism and decreases the synchronization delays between host and device.

In the example shown in Figure 4.2a, the unoptimized code repeats two sequences with different buffers. A computation is enqueued (lines 1 and 4), the result is read (lines 2 and 5) and cross-validated on the host (lines 3 and 6). The optimized version shown in Figure 4.2a simply moves the second kernel execution immediately after the first one (line 2). In this case, it has two advantages. First, both kernel invocations are now being side-by-side, exposing task parallelism – which was not available before since they were on either side of a blocking read. Second, computation on the host and the device overlaps: the first cross validation now proceeds while the device is computing the other output. In the original application, either the host or the device was idle at any point. This one liner produced a 42% performance boost.

However, this transformation requires knowledge about the dataflow between kernels: the code fragment does not specify the definitions of the kernels and their parameters. If the second kernel instance also modifies the buffer b1, then this transformation is not valid since it introduces a data race. As a consequence, while the actual modifications to the code are trivial, proving the validity of the transformation is difficult and requires a careful analysis of the dataflow and all the synchronization points.

```
kernel void MM(const global float* a,
               const global float* b,
                     global float* c,
               const int w, const int h){
  const uint
    off = get_global_id(0) / (w*h)*(w*h),
    row = get_global_id(0) % w,
    col = get_global_id(0) / w % h;
  float acc = 0.0f;
  for(uint k=0; k<w; ++k)
    acc += a[off+row+k*w]*b[off+k+col*w];
  c[get_global_id(0)] = acc;
}
```

(a) Unoptimized Device Code

```
kernel void MM(const global float* a,
               const global float* b,
                     global float* c,
               const int w, const int h) {
  const uint
    off = get_global_id(0) / 16*16,
    row = get_global_id(0) % 4,
    col = get_global_id(0) / 4 % 4;
  c[get_global_id(0)] =
    a[off+row+0] * b[off+0+col*4] +
    a[off+row+4] * b[off+1+col*4] +
    a[off+row+8] * b[off+2+col*4] +
    a[off+row+12] * b[off+3+col*4];
}
```

(b) Hand Optimized Device Code

Figure 4.3: Hand optimization propagating runtime constants. Here the generic matrix multiplication code has been specialized for 4×4 matrices.

4.2.2 *Code Specialization*

Code specialization integrates runtime knowledge about the execution context into the device code in order to increase performance. The specialization propagates information either about runtime constants or pointer analysis information.

CONSTANT PROPAGATION Scalar variables which are passed as kernel parameters can be replaced by their actual value prior to program compilation to optimize the device code. Constant values can be a very effective source of optimization, particularly if they determine part of the control flow, like loops or branches. Figure 4.3 shows a matrix-matrix multiplication application for two linearized matrix arrays; where each matrix of the first array is multiplied by the corresponding matrix in the second. The height and width of the matrices are unknown statically, so they are defined as kernel parameters. The optimized code presented in Figure 4.3b assumes the matrices have a dimension of 4×4. The code is specialized accordingly. In addition to simplifying the algebraic expressions, in this case the loop from the unoptimized code can be fully unrolled and the store can be done in-place. This optimization leads to a 1.53x speedup over the original code.

The specialized kernel, however, is only usable under very specific circumstances. If the application uses several matrix dimensions, it will require different specializations. This significantly increases code redundancy. Furthermore, the prototype of all the specialization is the same, making mistakes in the host code easy to make and hard to debug.

```
kernel void add(const global float* a,
                const global float* b,
                      global float* c) {
  const uint i = get_global_id(0);
  c[i] += a[i] + b[i];
}
```

(a) Unoptimized Device Code

```
kernel void add(const global float* a,
                const global float* b,
                      global float* c) {
  c[get_global_id(0)] *= 3;
}
```

(b) Hand Optimized Device Code

Figure 4.4: Hand optimization using knowledge about the host program to optimize aliasing memory accesses. Here the programmer assumes that the pointers a, b and c alias.

ALIAS ANALYSIS Programmers use knowledge about the runtime values to solve pointer aliasing. Compilers cannot eliminate redundant reads if the base pointers are different, and static analysis is helpless for unknown kernel parameters. Only specific knowledge about the runtime values can simplify these memory accesses. The *add* kernel in Figure 4.4 is a generic function where each element of an array is incremented by the sum of two other arrays. If a programmer knows that arrays a, b and c alias for a particular invocation, the kernel can be re-written as an in-place multiplication, resulting in two fewer load instructions. In order to isolate changes from the host code, the unused parameters can remain and will simply be ignored. This modification results in a 87% performance improvement.

While this optimization is effective, it is very difficult to apply in practice, since it leads to several versions of the kernel with identical signatures but different behaviors, and the OpenCL runtime cannot enforce the validity of the transformation.

4.2.3 *Kernel Fusion*

Manually fusing two or more kernels into one is a common optimization when there is either code or data access redundancy between the kernels. This is a very efficient optimization if the number of reused resources is high enough, since it reduces memory traffic, which is a frequent application bottleneck, especially on GPUs.

For this transformation to be applied, one must first cross reference host and device codes to identify the fusion candidates. The host code contains the dataflow paths linking allocated memory objects to computation and memory operations, and the device code contains information about how each individual input is used and whether it is read from or written to. Analyzing the dependency patterns can identify two categories of fusion candidates: data independent (or horizontal) fusion and data-dependent (or vertical) fusion.

```
// Device Code
kernel void WS(global float *w, global float *s)
{ const uint i = get_global_id(0);
  s[i]+=w[i]; }
kernel void WM(global float *w, global float *v,
               global float *m)
{ const uint i = get_global_id(0);
  m[i]+=w[i]*v[i]; }

// Host Code
cl::Kernel S{p,"WS"},M{p,"WM"};
S.setArg(0,w);S.setArg(1,s);
M.setArg(0,w);M.setArg(1,v);M.setArg(2,b);
queue.enqueueNDRangeKernel(S,{0},{g},{l});
queue.enqueueNDRangeKernel(M,{0},{g},{l});
```

(a) Unoptimized Code

```
// Device Code
kernel void WSM(global float *w,
                global float *s,
                global float *v,
                global float *m)
{ const uint i = get_global_id(0);
  s[i]+=w[i];
  m[i]+=w[i]*v[i]; }

// Host Code
cl::Kernel S{p,"WMS"};
S.setArg(0,w);S.setArg(1,s);
S.setArg(2,v);S.setArg(3,w);
queue.enqueueNDRangeKernel(S,{0},{g},{l});
```

(b) Hand Optimized Code

Figure 4.5: Example of data-independent (horizontal) fusion. The two kernels have distinct outputs, so their code can be merged to factorize the inputs.

HORIZONTAL FUSION Data independent fusion arises when the input and output sets of multiple kernels do not overlap. Since there is no dependency between outputs and inputs, the relative ordering of the operations does not matter, or they can execute simultaneously; hence they can be merged into a single invocation. The benefit of horizontal fusion depends on the amount of shared operations between the kernels being merged. Read operations to the same address are simplified to a single transaction, making it a very efficient transformation when the input sets overlap. Traditional compiler optimization passes can also simplify computation. Similar loops and branches can be fused to reduce the control flow graph and decrease the number of predicate evaluations; common expressions can be eliminated. Code vectorization also has greater potential on larger kernels.

Figure 4.5 shows a simple example of data independent kernel fusion. A weighted sum of multiple arrays is computed using two kernels. The first sums the weights and the second sums the weighted contribution. Both kernels use the same weight array but they write to two different arrays representing the total weight and total contributions. Since the two outputs are disjoint, the two kernels can be merged into one. In the fused kernel, the weight is read only once by each thread instead of twice, improving performance by 71%.

This is a heavy transformation. It requires the implementation of a new kernel, and a modification of all the kernel invocations using this particular call sequence in the host code. It introduces redundancies in the device program if both original and fused versions are used. The maintainability and reusability of the code is also greatly impaired.

```
// Device Code
kernel void AX(const float a,
               global float *x)
{ x[get_global_id(0)] *= a; }

kernel void PY(global float *x,
               const global float *y)
{ const uint i = get_global_id(0);
  x[i] += y[i]; }

// Host Code
cl::Kernel AX{p, "AX"},PY{p, "PY"};
AX.setArg(0,a); AX.setArg(1,b1);
PY.setArg(0,b1); PY.setArg(1,b2);
queue.enqueueNDRangeKernel(AX,{0},{g},{l});
queue.enqueueNDRangeKernel(PY,{0},{g},{l});
```

(a) Unoptimized Code

```
// Device Code
kernel void AXPY(const float a,
                 global float *x,
                 const global float *y){
  const uint i = get_global_id(0);
  x[i] = x[i] * a + y[i];
}

// Host Code
cl::Kernel AXPY{p, "AXPY"};
AXPY.setArg(0,a); AXPY.setArg(1,b1);
AXPY.setArg(2,b2);
queue.enqueueNDRangeKernel(AXPY,{0},{g},{l});
```

(b) Hand Optimized Code

Figure 4.6: Example of data-dependent (vertical) fusion. The ouput of the first kernel *AX* is used as input of a second kernel *PY* Each kernel load and store the same element. Merging them into a single kernel *AXPY* performs the same computation but reduces the number of memory transactions by half.

VERTICAL FUSION Vertical fusion results from detecting particular algorithmic patterns which guarantees the safety of the transformation. When the output of a kernel is consumed as an input of another kernel, it might be beneficial to merge the kernels and avoid using the global memory to store the temporary values. However, the lack of global synchronization in OpenCL means that this transformation is only valid if the data is produced and consumed within the same thread or workgroup. This is only known by combining knowledge about the kernel arguments and the domain from the host application, and the memory access patterns from the device code.

In the example shown in Figure 4.6, it is easy to verify this assumption: the kernel AX stores the value at address *global_id(0)* and the kernel PY reads from an identical offset. An inspection of the host code shows that both kernels are invoked within the exact same range, hence each store from the first kernel exactly corresponds to a load in the second kernel. Because there are no other dependencies, the two kernels can be fused into one. The resulting kernel contains one fewer load from x and one fewer store. Removing the temporary array altogether is very effective in this case, with a speedup of 2.11x.

Like horizontal fusion, this is a very intrusive transformation which requires extensive refactoring of the host and device code. Maintainability and reusability of the kernels is also considerably decreased, and code duplication increases.

```
// device code
kernel void DBL(global float *x,
                global float *y)
{ const int gid = get_global_id(0);
  x[gid] = 2*y[gid]; }

// host code
cl_mem b1 = /*...*/, b2 = /*...*/;
cl::Kernel DBL = /*...*/;
DBL.setArg(0,b1); DBL.setArg(1,b2);
q.enqueueNDRangeKernel(DBL,{0},{g},{l});
q.enqueueNDRangeKernel(DBL,{0},{g},{l});
q.enqueueReadBuffer(b2,true,0,size,res_b2);
q.enqueueNDRangeKernel(DBL,{0},{g},{l});
clReleaseMemObject(b2);
```

(a) Unoptimized Code

```
// device code
kernel void DBL(global float *x,
                global float *y)
{ const int gid = get_global_id(0);
  x[gid] = 2*y[gid]; }

// host code
cl_mem b1 = /*...*/, b2 = /*...*/;
cl::Kernel DBL = /*...*/;
DBL.setArg(0,b1); DBL.setArg(1,b2);
q.enqueueNDRangeKernel(DBL,{0},{g},{l});
q.enqueueReadBuffer(b2,true,0,size,res_b2);
clReleaseMemObject(b2);
```

(b) Hand Optimized Code

Figure 4.7: Example of task elimination. Computation is sometimes performed needlessly when the output is overridden or released. These artifacts are hard to find in complex code since the required information is scattered.

4.2.4 *Task Elimination*

Knowing the complete dataflow path of an application by tracking the lifetime and usage of allocated memory allows for some simple yet efficient optimizations. For example, a kernel invocation for which the output is never used by any subsequent invocations nor read from the host program is an obvious waste of compute resources. However, these are hard to find in practice, since memory management and computation are scattered in the source code and tracing the dataflow paths is complicated. Moreover, these dead invocations might arise as an artifact of the development or optimization processes in a highly complex code, like reading temporary buffers in the host code for debugging purposes or enqueueing unecessary operations in command batches.

Figure 4.7 shows an example containing two dead tasks. Spotting them requires cross-referencing the kernel arguments, the iteration domains, the lifetime of the objects, the other commands, the synchronization points and the device code – all at once and for each command. The first instance is immediately overwritten by the second one since all the aforementioned properties are equal. The last instance is also never used and the buffer is immediately released. Both of these instances can be removed in the optimized version.

Note that this is an extremely precarious transformation. For example, changing the kernel to write to the first argument would invalidate the first deletion. Also, since memory

objects are explicitly reference counted, retaining b2 anywhere in the code would break the second deletion because the buffer would still be reachable.

A more fine-grained version of this transformation consists of removing dead stores in a kernel invocation. This is applicable whenever a kernel has multiple outputs, but a subset of these is never subsequently read. In this case, the kernel invocation cannot be eliminated, but the dead store operations can be removed by specializing the kernel. This process is similar to the alias analysis specialization described earlier.

4.2.5 *Transformation Applicability*

The dynamic optimizations presented in this section increase the performance of a sequence of commands in OpenCL. They are all intrusive and require modifications of the host or device code, or even both. The source code resulting from the optimization process is precarious, since it relies on a set of assumptions defined by the programmer and not enforced by the runtime. It often leads to code duplication, where the optimized variant lives alongside the generic version – one providing performance and the other portability if the specialization is not applicable. This decreases the overall maintainability of the code.

In isolation, all these optimizations have typically shown a roughly equal performance benefit. However, on much more complex applications, the best optimization set is not well-defined. Implementing them incrementally using an arbitrary ordering is far from optimal and requires a constant refactoring of the same code. The complexity of the optimization also increases everytime the code is modified. For instance, fusing two kernels which have several specializations each requires implementing all specialization combinations by hand.

Moreover, since these optimizations affect different types of bottlenecks (as shown in Table 4.1), the optimization process is often subjective and only considers the particular set of architectures used for development, decreasing the performance portability of the code.

To replace fragile hand optimizations, we present Helium, a transparent OpenCL optimizer which applies all the transformations presented in this section automatically and dynamically. Helium traces an application at runtime, finds the minimal set of dependencies between OpenCL actions and optimizes the task sequence before it is transparently executed by the vendor implementation. New kernels are created on the fly from the initial set of kernels to improve data locality, and tasks are parallelized automatically to take advantage of the task parallelism provided by the OpenCL model.

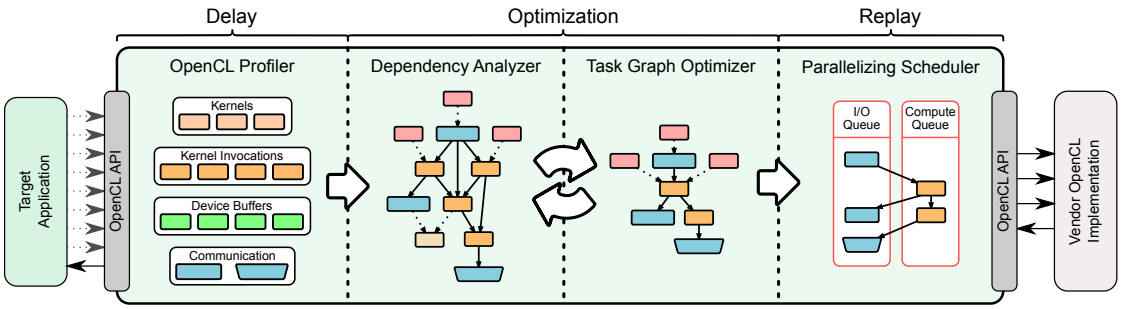


Figure 4.8: Overview of HELIUM. OpenCL function calls from a target application are intercepted to gather profile information. This information is then analyzed and combined to build a task graph, which is optimized before being executed by the vendor implementation.

4.3 HELIUM OPTIMIZER OVERVIEW

The HELIUM optimizer uses a *delay-optimize-replay* mechanism; all non-blocking OpenCL commands are postponed and executed lazily in order to gather and exploit as much runtime information as possible and depict a broader execution plan spanning multiple kernel invocations. The optimizer then improves these execution plans before replaying them.

Figure 4.8 shows an overview of the framework and its mechanism. A target application invokes functions from the OpenCL API. These calls are intercepted by the framework, which dynamically optimizes the execution flow before forwarding the calls to the vendor implementation. Because this process takes place between the host program and the vendor implementation, it can be deployed transparently over any existing application without any code refactoring. The main steps of HELIUM's internal optimization process are:

1. *Delay*: each OpenCL function invoked by the host application is silently intercepted and transparently converted into a self-contained asynchronous task.
2. *Analysis*: The tasks are linked together to create a task graph. Their dependencies are minimized by combining static device code and host runtime analyses.
3. *Optimization*: The task graph is then optimized dynamically in order to improve the performance of a sequence of operations. The transformations described in Section 4.2 are automated and a guided optimization process iteratively applies them.
4. *Scheduling*: the task graph is *replayed* in topological order using a parallelizing scheduler, exploiting task parallelism. The host program is blocked until completion of all the actions required to restore consistency in the target application.

The following section describes the implementation of these components in more detail.

4.4 HELIUM IMPLEMENTATION

This section presents the implementation of the general principles behind Helium. The delay mechanism is first described in Section 4.4.1; the soundness of the technique is demonstrated by analyzing the OpenCL model and showing that enough information can be extracted to guarantee the safety of a delayed execution for a sequence of OpenCL commands. This phase produces a task graph which is then optimized by combining static and runtime information and applying the transformations described in Section 4.4.2. Finally the optimized schedule can be executed by the vendor implementation using the scheduler presented in Section 4.4.3.

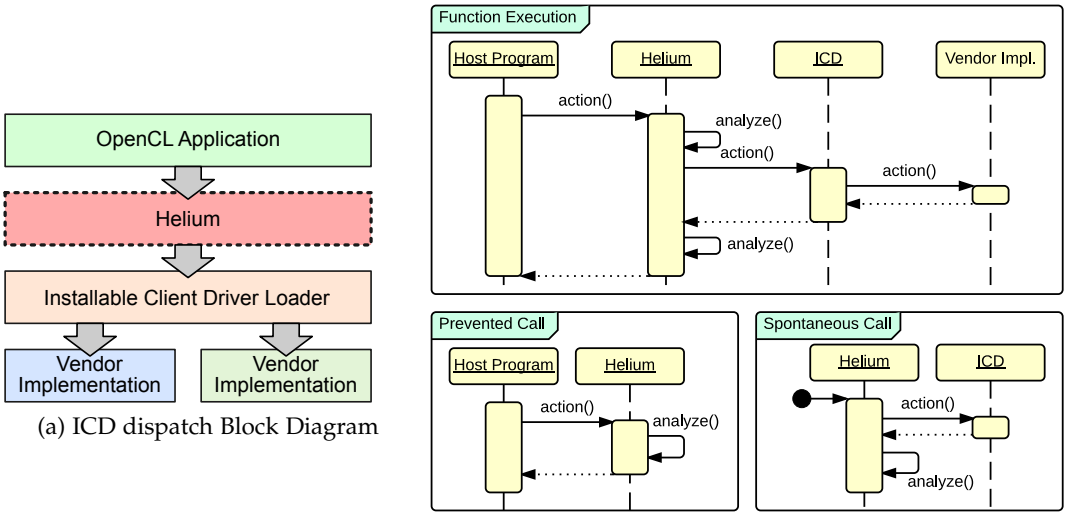
4.4.1 *Delay and Analysis*

The delay mechanism allows Helium to be transparently interposed between the host application and the vendor implementation. We present here the techniques used to intercept the calls from the host application and create a synthetic OpenCL execution context within Helium to force the host program to process as far as possible before synchronizing with the device. This process creates an asynchronous task graph which is critical to implement the optimizations described in the following subsection.

4.4.1.1 *Intercepting OpenCL API Calls*

Conforming OpenCL implementations must define all the API functions and provide a set of additional functions interfacing with the Khronos Installable Client Driver (ICD). This enables multiple implementations of the same library to co-exist on a single system using an common layer called an ICD loader. The ICD loader gathers all the available platforms from all vendors using a dynamic linker API, and presents them to the host application. Once the host has chosen a platform, the subsequent OpenCL calls are forwarded by the ICD loader to the corresponding vendor implementation using a dispatch table. Because it is a very thin layer with a simple indirection, the ICD loader and the vendor implementation are referred to interchangeably in the remainder of this section.

Helium also implements the full OpenCL API but does not rely on the ICD loader. Instead, it is explicitly pre-loaded when executing an application. This adds an additional



(a) ICD dispatch Block Diagram

(b) Normal and alternative call sequences

Figure 4.9: OpenCL function call dispatch. The vendor implementations are managed by an ICD loader to provide a unified interface to the client application. Helium is inserted between the application and the loader (a) and takes control of the call forwarding (b).

layer between the host application and the loader, as shown in Figure 4.9a. The function execution use case in Figure 4.9b shows the modified sequence of an OpenCL call. It is intercepted before it reaches the loader and HELIUM takes control of its execution or can insert any callback. By dispatching the calls to the ICD loader, HELIUM can invoke the underlying vendor implementation to execute the device program.

HELIUM reserves the right to prevent or alter any OpenCL call from the running application or initiate spontaneous calls to the vendor implementation at any time, as long as the program semantics are preserved. These guarantees are provided by the analysis described in this and the following subsections.

4.4.1.2 *Static Analysis of the device code*

To take full advantage of the portability offered by OpenCL, most applications provide the device program as OpenCL-C source code, which is then compiled at runtime for the targeted device using the compiler provided by the selected vendor. This process is triggered explicitly through the API in a three-stage process where the source code is first registered (*clCreateProgramWithSource*), compiled (*clCompileProgram*) and linked (*clLinkProgram*) or directly built (*clBuildProgram*). HELIUM intercepts these calls to initiate a compilation in its embedded OpenCL-C compiler.

```
1  kernel
2  void MM(global float* C,
3          global float* A,
4          global float* B,
5          int wA, int wB){
6   int x=get_global_id(0); // gid_0
7   int y=get_global_id(1); // gid_1
8
9   float value=0;
10  for(int k=0;k<wA;++k)    // loop_1({0,+,1}_k^{wA-1})
11  {
12   float a=A[y*wA+k];      //   LD_glob({{(sizeof_float * wA * gid_1 + A),+,sizeof_float}^{wA-1})
13   float b=B[k*wB+x];      //   LD_glob({{(sizeof_float * gid_0 + B),+,(sizeof_float * wB)}^{wA-1})
14   value += a*b;
15  }                        // end_loop_1
16
17  C[y*wA+x] = value;       // ST_glob(sizeof_float * (gid_0 + gid_1 * wA) + C)
18 }
```

Figure 4.10: Static device code analysis: the source code of the device application (left) is intercepted by HELIUM and compiled in its internal compiler. An analysis pass examines pointer usage and generates partially evaluated PADs expressions for each memory access (right).

MEMORY ACCESS ANALYSIS The code is analyzed to gather characteristics of the kernels which are later used by the runtime to drive the optimizations. The source code is first compiled to an intermediate representation and multiple optimization and analysis passes are applied. Each memory access is examined, and a Pointer Access Description (PAD) representation [EG01] is generated. By following the control flow of the program and interpreting arithmetic instructions, a partially evaluated expression can be incrementally built up until it is used by a load or store instruction. For example, the last line of the program in Figure 4.10 is an assignment to $C[y * wA + x]$, and each of component in this expression can be traced across the kernel. In this case, C and wA are kernel parameters, and x and y are coordinates in the index space assigned to the global index at dimensions 0 and 1 respectively. The analyzer performs a context-sensitive analysis and can translate the call to builtin OpenCL functions to special markers in the expression, here gid_0 and gid_1 symbolize the global index. The store is to global memory since the pointer was annotated with this address space, so it is noted as ST_{glob}. The datatypes and their size can also be deduced from the kernel prototype. Putting everything together gives the following representation for this particular store:

$$\underbrace{ST_{\mathrm{glob}}}_{\text{Global Store}}\ \underbrace{(\overbrace{\mathrm{sizeof}_{\mathrm{float}}}^{\text{Type size}} * (\overbrace{\mathrm{gid}_0 + \mathrm{gid}_1}^{\text{Index Space}} * \overbrace{wA}^{\text{Parameters}}) + \underbrace{C}_{\text{Base}})}_{\text{Offset}}$$

LOOP ANALYSIS A slightly more complex representation is used for partially evaluated loops. PAD uses chains of recurrences (Chrec) to represent a polynomial or exponential function over a multidimensional index space [Zim93; Zim95]. Their general form is $\{init, \odot, stride\}$, where *init* is the initial value when entering the loop, *stride* is a loop invariant value iteratively applied to *init* using the operator $\odot$. For example, a loop computing the affine function $2 * i + 10$, where i is the iteration count, can be noted as $\{10, +, 2\}_i$.

Most compilers use this analysis to optimize loops with unknown or large trip counts by applying loop strength reduction or loop restructuring. However, static compilers cannot use unknown loop trip count for their transformations, and the analysis is discarded. On the other hand, HELIUM preserves the analysis and builds expressions for backedge evaluation. We introduce the additional superscript notation *trip*, where trip is a partially evaluated loop trip count. In Figure 4.10, the *for* statement at line 10 is a simple loop of k from 0 to wA, with an increment of 1 and a trip count of $wA - 1$, noted $\{0, +, 1\}_k^{wA-1}$. Inside this loop, the load $B[k * wB + x]$ in line 13 can be expressed using the previously described notation as:

$$(k * sizeof_{float} * wB + gid_0 * sizeof_{float} + B)$$

This can be combined to the loop notation by adding the loop independent values to the initial value of the Chrec notation and multiplying the loop dependent expression by the stride to give the final notation:

$$\underbrace{LD_{glob}}_{\text{Global Load}} \left(\{ \underbrace{(sizeof_{float} * gid_0 + B)}_{\text{Start Value}}, \underbrace{+, (sizeof_{float} * wB)}_{\text{Stride}} \}\right) \underbrace{^{wA-1}}_{\text{Trip Count}}$$

These expressions are a compact form for collecting information for each kernel to represent its properties. For example, if a pointer appears in an *LD* PAD and none of the *ST* PADs, like A and B, then it is read-only. Likewise, write-only, read-write and no-access can be easily derived, as well as an estimate of the number of accesses. HELIUM builds a lookup table of all PADs for each compiled kernel when the device code is compiled and retains it throughout the program execution.

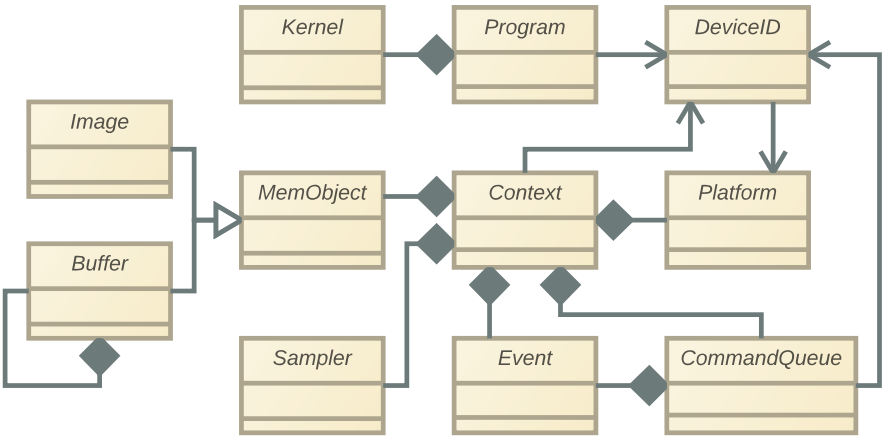


Figure 4.11: Overview of the OpenCL vendor objects and their relationships as described in the standard. The standard does not specify the definition of these objects, the API manipulates them using opaque pointer types called object handles.

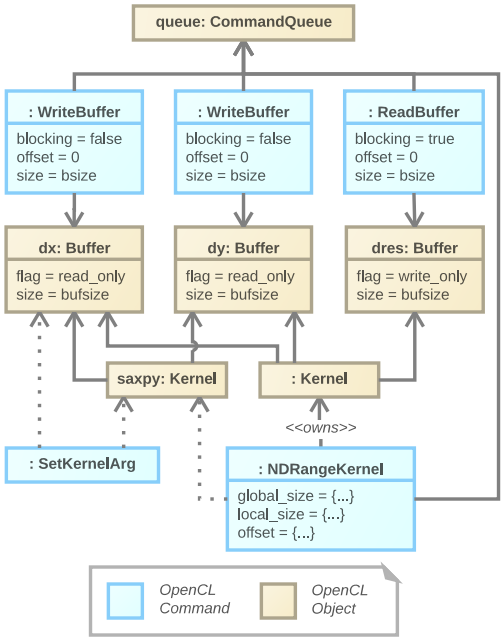
4.4.1.3 *OpenCL Runtime Analysis*

Helium gathers high-level information from observing the interaction between the application and the vendor implementation. This execution profile is used to derive properties which validate the safety of the delay mechanism presented later in this section. For the sake of simplicity, the analysis mechanism described in this subsection assumes that every call from the host application intercepted by Helium is immediately executed by the vendor implementation. This behavior is represented by the function execution case in Figure 4.9b, which is the same functional behavior as if the application was running without Helium. When the call returns from the implementation, it goes through Helium again and at this point information about the OpenCL execution context can be gathered to mirror the internal state of the vendor implementation. The following subsection uses this analysis as a foundation of the delay mechanism.

OBJECTS & HANDLES Every OpenCL implementation is based on a set of objects representing entities in the OpenCL model. The relationships between these objects, presented in Figure 4.11, are provided by the OpenCL standard as well as a set of properties that they must contain. Their definition, however, is vendor specific. To decouple the API and the vendor implementation, the underlying objects are obfuscated from the users who only manipulate opaque pointers in the OpenCL API, called *object handles*. However, accessing the underlying objects is essential for understanding the execution context of the application and how commands relate to each other. Hence Helium has to re-create this context.

```
1  cl_context ctx=/*...*/;
2  cl_command_queue queue=/* in order queue*/;
3  cl_program prog=/* ... */;
4  cl_kernel saxpy=clCreateKernel(
5      prog,"add",NULL);
6  cl_mem dx=clCreateBuffer(
7      ctx,CL_MEM_READ_ONLY,bufsize,NULL,NULL);
8  cl_mem dy=clCreateBuffer(
9      ctx,CL_MEM_READ_ONLY,bufsize,NULL,NULL);
10 cl_mem res=clCreateBuffer(
11     ctx,CL_MEM_WRITE_ONLY,bufsize,NULL,NULL);
12
13 clEnqueueWriteBuffer(
14     queue,dx,CL_FALSE,0,bsize,x,0,NULL,NULL);
15 clEnqueueWriteBuffer(
16     queue,dy,CL_FALSE,0,bsize,y,0,NULL,NULL);
17
18 clSetKernelArg(saxpy,0,sizeof(cl_mem),&dx);
19 clSetKernelArg(saxpy,1,sizeof(cl_mem),&dy);
20 clSetKernelArg(saxpy,2,sizeof(cl_mem),&res);
21
22 clEnqueueNDRangeKernel(
23     queue,saxpy,1,NULL,&glb,NULL,0,NULL,NULL);
24 clSetKernelArg(saxpy,2,sizeof(cl_mem),&dx);
25 clEnqueueReadBuffer(
26     queue,dy,CL_TRUE,0,bsize,y,0,NULL,NULL);
```

(a) Input Code



(b) OpenCL Context after execution

Figure 4.12: Runtime Analysis of the OpenCL Context: by intercepting all API call in an application, HELIUM can re-create an internal view of the OpenCL context and establish relationships between Object Handles and Commands.

The API is used to instantiate OpenCL entities implicitly, change their properties or create relationships. However, the information is sparse and difficult to recover. Figure 4.12a shows an example of application creating three buffers and computing the sum of the first two into the third array. In isolation, any command carries very little information about its execution context. For example, the kernel invocation at line 22 only uses two handles: one for the queue and one for the kernel. However, after reading the entire program, it is clear that the kernel uses some memory buffers, which are also used in other parts of the application. Knowing the relations between this command and the surrounding memory operations requires a deeper analysis.

OBJECT ASSOCIATIONS HELIUM implements the same model as the vendor implementation and maintains the same set of objects and their state throughout the program execution. Objects are instantiated explicitly – generally when invoking a *clCreate** function –, and tracked using their handle. Specific functions create associations between objects: for example, setting a kernel argument binds a memory object to a kernel argument, as shown in lines 18-20. Similarly, event objects can be linked to commands; and commands are as-

sociated to queues. This creates a network of interconnected objects and commands with their properties, capturing the entire execution context. This is visualized in Figure 4.12b.

COMMAND DEPENDENCIES The relationships between the objects can in turn be used to infer the dependencies between asynchronous command queue actions. This is used to detect *order independent commands*, which are commands that could be re-ordered without any observable effect on the device program – since they do not interfere with each other. This analysis is objective and autonomous from the explicit dependencies provided by the user. For instance, using an in-order queue creates explicit temporal dependencies between every commands. However, these dependencies are usually over-specified since some commands might actually be independent. By contrast, the object-flow approach provides fine-grained minimized dependencies.

If two command queue actions are pushed in the same queue and use unrelated objects (except from sharing a common OpenCL context, as they must for the program to be correct), they are order-independent. This is valid since the model guarantees strong isolation between the different entities: an object cannot possibly alter the state of any other object unless they have been explicitly bound through the API. In the example of Figure 4.12a, the first two writes are independent since they use distinct and unrelated memory objects. Hence the order of the two commands could be swapped without compromising program semantics. This proves that the temporal dependency introduced by the in-order queue is not necessary for correct execution and too restrictive.

Conversely, if any two commands share related objects, there is a dependency determined by the order they were pushed in the queue. Note that this can be an indirect association. In the example, the kernel instance uses the kernel object, which has been associated to the same two buffers objects used for the write commands. Hence the write operations have to be executed before the kernel.

TEMPORAL CONSISTENCY For this technique to be robust over time, the context of each command must be the same as it was when it was delayed. This presents a problem since objects might change during an asynchronous execution. In the example, the third argument of the kernel is changed at line 24, which is after its instantiation but before the read is enqueued at line 25. By the time the context is examined for the read command, the

kernel is no longer associated with the buffer being read, thereby missing a crucial ordering dependency. To avoid introducing these data races, the full context of each command must be preserved.

This is done by copying the objects associated with the command: the original objects represent the up-to-date context, and the private copies store the properties of the objects at the time the command was instantiated. HELIUM distinguishes between immutable objects, which cannot be modified through the OpenCL API, and the mutable objects, which can. Memory objects are immutable and hence are not copied, but kernel objects are mutable so the instantiation creates a private copy of the state of the kernel and its associations, as shown in Figure 4.12b. This re-establishes the dependency between the read command and the kernel execution, avoiding the data race.

This analysis shows that the dependencies between a chain of asynchronous commands in an arbitrary OpenCL application can be derived from observing the interactions between the application and the vendor implementation. This dependency specification is more fine grained than what the user originally expressed, allowing the detection of time independent commands, which can safely be reordered. The state of the OpenCL model objects at any point during the execution is known and a representation of any command context can be stored. The next section introduces the delay mechanism, which takes advantage of these observations to create delayed task graphs transparently.

4.4.1.4 *Delay Implementation*

The delay mechanism decouples the host and device programs and introduces command batching without compromising program semantics. In other words: the host application believes all issued commands are getting executed by the vendor implementation, even when they are not; and the vendor implementation will eventually execute something semantically equivalent – but not necessarily identical – to the intended execution. This process is undetectable from the application and from the vendor implementation, but opens a new scope for collective optimizations.

The key underlying concept consists of creating two isolated execution contexts and maintaining two copies of the OpenCL objects: one representing the view of the OpenCL objects as intended by the host application and the other containing the profiling information of the actual execution in the vendor implementation. HELIUM can manipulate the two

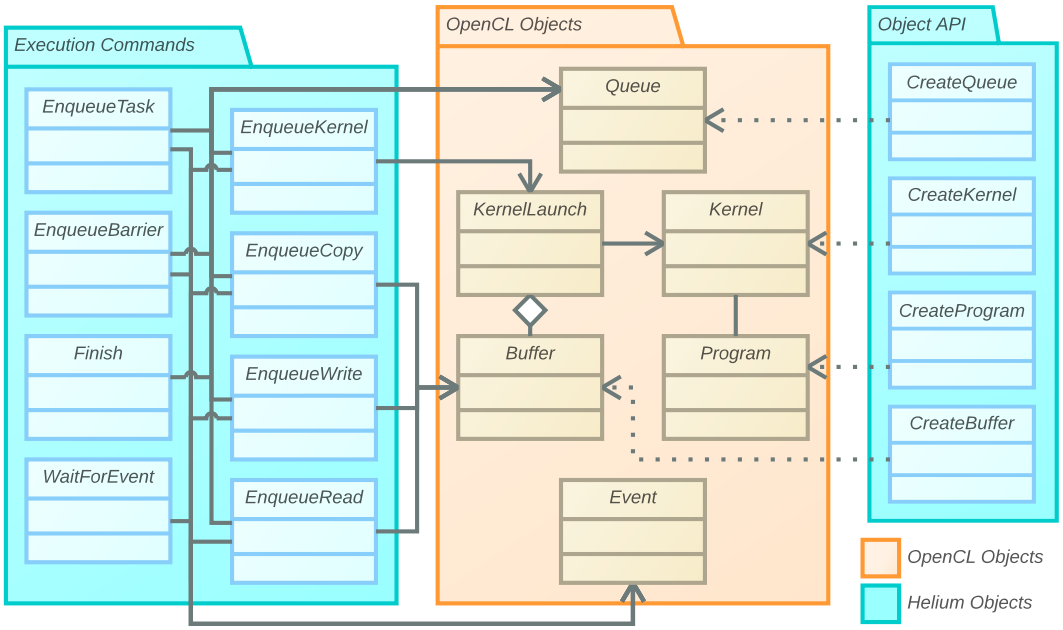
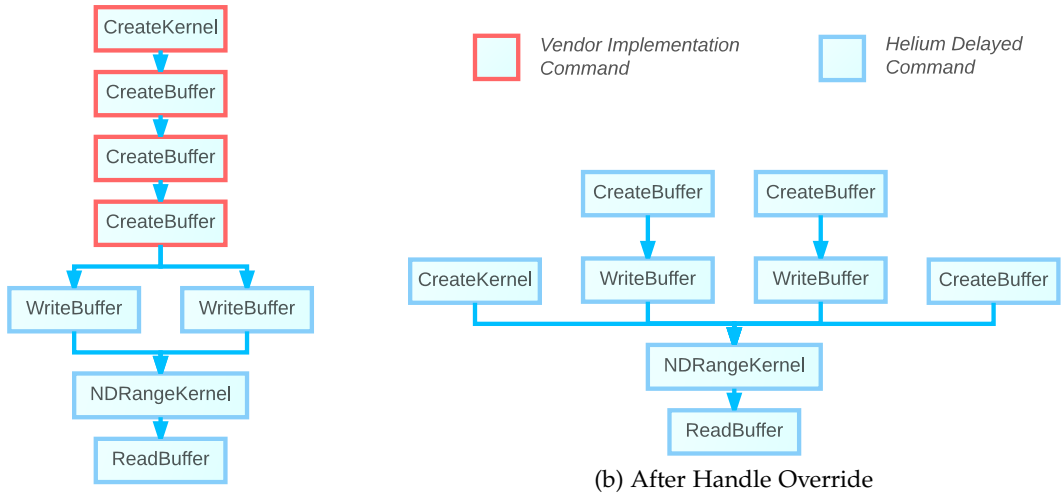


Figure 4.13: Overview of Helium's command objects and their relation to OpenCL objects. The delay phase consists of generating functor objects wrapping OpenCL commands instead of invoking the vendor implementation.

views independently as long as it provides two guarantees: the first is that Helium creates a synthetic observable context for the host application which is identical to what it would have been if the vendor implementation executed the commands; and the second is that synchronization points between the host and the device programs restore a fully consistent state in the host application. These guarantees are defined and described below.

FUNCTOR OBJECTS When the host application calls an OpenCL function, Helium intercepts it and generates an object representation of the command. A set of functor objects are defined to represent each API function and integrated as an extension of the OpenCL objects. The main functor objects and their relations to the OpenCL objects are presented in Figure 4.13. Each function call triggers the instantiation of the corresponding functor and the call parameters are deeply copied. Together with its complete execution context, the command can be executed independently from the host application.

The functor is pushed in an internal command queue, and the call is not forwarded to the vendor implementation: it is a *delayed command*. The obvious issue is that return values, which depended on the vendor implementation, are undefined but necessary for the host program to execute correctly. Hence, Helium has to create a synthetic context to replace the vendor implementation.



(a) After Error Code Override

(b) After Handle Override

Figure 4.14: Details of the delay process: to force an synchronous evaluation of OpenCL commands, HELIUM can override the return code of queue commands (a), or override all the OpenCL Handles (b) without any impact on the host application.

VIRTUAL CONTEXTS Most commands in OpenCL are asynchronous and have limited immediate side effect on the host application. The only two synchronous interactions are error codes - which are either returned or stored in pointer arguments –, and object handles. For Helium to be undetectable, is has to preserve the observable behavior for both.

Since HELIUM assumes the commands will always execute successfully[1], it overrides the error code to the success flag for the delayed commands. This enables all asynchronous command queue actions to be freely reordered until another OpenCL function is called, which is the task graph presented in Figure 4.14a.

This is still fairly limiting since asynchronous commands are often interleaved with other OpenCL commands which modify the context, like kernel object creation or buffer allocation. These functions return OpenCL handles, which are used to define relationships between other commands. The previous section described how HELIUM can track these objects to build a valid and complete execution context, capturing all relationships between them. The delay mechanism uses this internal model in the same way a vendor implementation would and overrides the returned OpenCL handles with pointers to its internal representation.

HELIUM can then delay any command in the OpenCL API while preserving enough information to guarantee a correct execution when they are later evaluated. As shown in Figure 4.14b, all commands from the code example are delayed and none of them were

1 Implications of faulty commands, error handling and other side effects are discussed in Section 4.5.

actually executed by the vendor implementation. The buffers are not allocated and nothing is pushed in the queue. However, HELIUM captured all these commands and produced the same context from the host application point of view.

Commands cannot be delayed indefinitely, since the host application might use the output of the device. The final stage of the delay mechanism is to identify when commands have to be executed. Creating so-called *synchronization points*.

SYNCHRONIZATION POINTS Any computation offloaded to the device in OpenCL, and most of the communication, is executed asynchronously; allowing host and device programs to make independent progress in parallel. However, this model requires the host application to explicitly initiate synchronization with the device before it can safely acquire the state of an asynchronous output. OpenCL provides many ways of synchronizing host and device: users can either wait for all pending actions in a queue to execute (*clFinish*), or a specific set of asynchronous tasks (*clWaitForEvents*), or enqueue a blocking memory operation, which guarantee all previous operations or its user-defined dependencies will also complete. These synchronizations are necessary to ensure correctness of the program, but in the same way in-order queues are often introducing over-specified dependencies, user defined synchronization is often overly conservative. This decreases performance by preventing host and device programs from executing in parallel and impairs the applicability of inter-command optimizations by generating short task graphs. Fortunately, HELIUM can override user-specified synchronization to enforce it only when strictly needed.

Although it has not been formally proven, synchronization is only necessary in two cases: to avoid data races in multiple unsequenced kernels or to acquire and release memory states in the host application. Since the analysis performed in HELIUM can automatically detect data races between asynchronous kernel executions, as explained earlier, this leaves only acquire and release operations. These are characterized by a pending operation involving any exchange between the host application memory space and the OpenCL managed memory space, followed by an explicit user-defined synchronization request. At this point, because HELIUM cannot know how the host application uses the data in its own memory space, the command must be executed to restore coherency. In the example in Figure 4.12a, the write operations satisfy half of the requirements: they involve a pointer from the host application, but they are asynchronous, so they can still be delayed. The final read, however,

is blocking. Therefore it becomes a true synchronization point. The consistency between the host and the device program must be restored at that point, since the host application is free to use the data at any point after this operation.

HELIUM pauses the execution of the host program when a synchronization point is encountered. The coherency between the host and the device applications can be restored by executing the list of delayed commands. The function then returns after coherency is restored, and the cycle restarts until the next synchronization point is reached. This introduces command batching where all commands between synchronization points are collected and their execution delayed without corrupting the host program.

4.4.1.5 *Delay Phase Summary*

This section presented the delay mechanism implemented in HELIUM. HELIUM can transparently and safely prevent OpenCL function calls from being immediately executed by the vendor implementation. The actions are instead turned into self-contained functor objects, creating a task graph which can be executed in a delayed fashion without any impact on the host application. This allows HELIUM to take full control of *clCreate** and *clEnqueue** functions. The dependencies between commands are deduced by combining their basic semantic information and their used of object handles. The synchronization points are automatically detected and interrupt the delay mechanism to restore consistency. Before being executed, the task graphs are sent to HELIUM's optimizer.

4.4.2 *Task Graph Optimizer*

The previous subsection demonstrated that an OpenCL application can be safely decoupled from the vendor implementation by encapsulating and delaying commands until a synchronization point is reached. We also described how HELIUM uses its embedded compiler to gather static information about the device code. The information gathered is very incomplete on both sides: command dependencies in the generated task graph are overly cautious since the runtime does not specify how the data is used and the static analysis of the device code provides little information on its own. The task graph optimizer combines both the runtime and static information to optimize the edges and the nodes in the graph.

```
1  int n = computeN(); // 'n' is computed at runtime
2
3  // Compile device source code
4  std::string c = R"(
5   #define ID get_global_id(0)
6   #define buf global int*
7   kernel void A(buf a, buf b)
8      { b[ID] = 2 * a[ID]; }
9   kernel void B(buf t, buf u)
10     { u[ID] = t[ID-1] - 1; }
11  kernel void C(buf x, buf y, buf z, int n)
12     { y[ID+n] += x[ID-n] + z[ID-n]; } )";
13 Program p{cx,{{c.data(),c.size()}}}; p.build(devs);
14
15 // Memory allocation
16 auto mf = CL_MEM_READ_WRITE;
17 Buffer b1{cx,mf,siz},b2{ctx,mf,siz},b3{ctx,mf,siz};
18
19 // Kernel creation and argument binding
20 Kernel A{p,"A"}; A.setArgs(b1,b2);
21 Kernel B{p,"B"}; B.setArgs(b1,b3);
22 Kernel C{p,"C"}; C.setArgs(b2,b1,b3,n);
23
24 // Enqueue non-blocking operations
25 q.enqueueWriteBuffer(b1,CL_FALSE,0,siz,data);
26 q.enqueueNDRangeKernel(A,{0},{100},{1});
27 q.enqueueNDRangeKernel(B,{1},{100},{1});
28 if( n > 1 ) { // dynamic computation
29    q.enqueueReadBuffer(b2,CL_FALSE,0,siz,tmp);
30    q.enqueueNDRangeKernel(C,{n},{100-n},{1});
31    n = 0;
32    C.setArg(3,n); C.setArg(0,b3); A.setArg(0,b2);
33 }
34 q.enqueueNDRangeKernel(A,{0},{100},{1});
35 q.enqueueNDRangeKernel(C,{n},{100-n},{1});
36
37 // Blocking operation
38 q.enqueueReadBuffer(b1,CL_TRUE,0,siz,result);
39 if( n > 1 ) { b2 = Buffer(); }
```

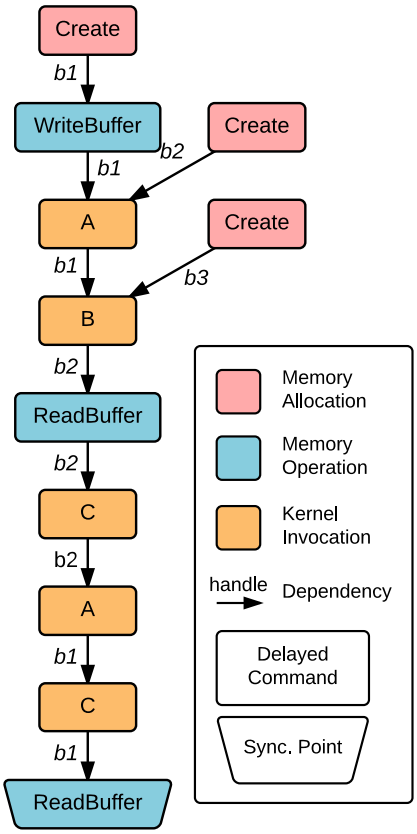


Figure 4.15: Example of dynamic multi-kernel application and the associated task graph generated by HELIUM's delay mechanism for the value $n = 2$.

The example shown in Figure 4.15 will be used throughout the section to demonstrate the optimizations. It contains seven asynchronous commands, two of which are enqueued conditionally, and one blocking read at line 38. HELIUM delays all commands using the mechanism described in Section 4.4.1, which produces the task graph shown on the right-hand side. Because all commands use the same pool of objects, they are inter-dependent. However, it can be improved, first by optimizing the dependencies between the nodes and then the nodes themselves.

4.4.2.1 *Dependency Optimizations*

The dependency optimizer deletes or creates edges in the task graph to represent the minimal set of data-dependency between the nodes. It uses a much more fine-grained approach

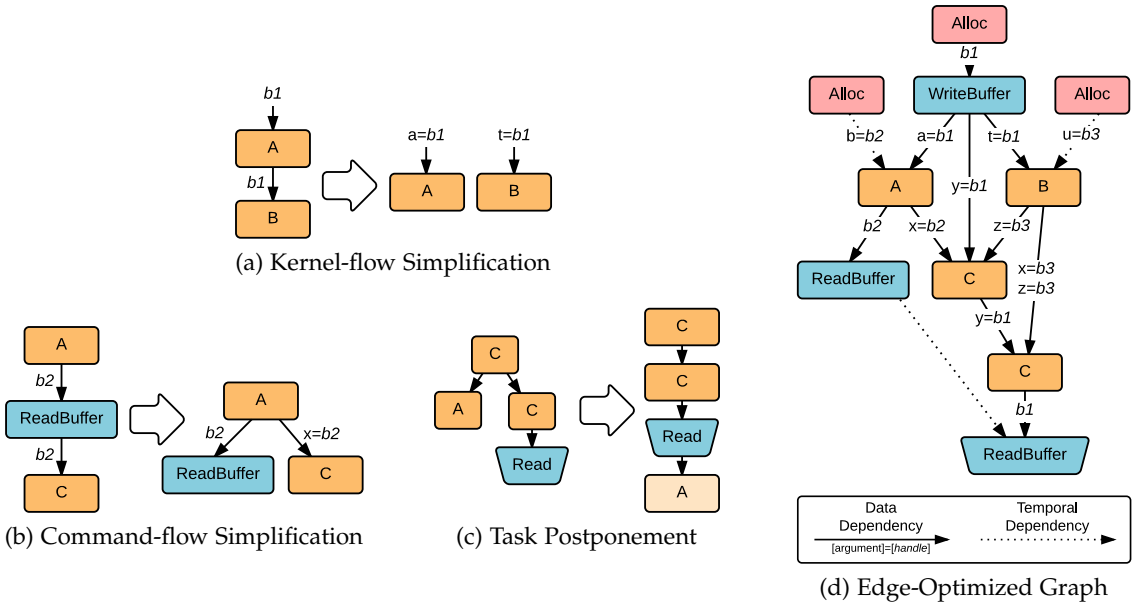


Figure 4.16: Task Graph Edge Optimization: HELIUM uses the device code analysis to break unnecessary dependencies between kernel instances (a) or with other commands (b). It can also create additional dependencies to enforce a particular ordering between the commands (c). The resulting task graph (c) is much wider and exposes task parallelism.

than the delay mechanism and inspects how each memory object is used throughout the graph and where it can be shared by multiple actions instead of assuming each command takes ownership of the object. This results in a wider graph, presented in Figure 4.16d. Each edge optimization is described below.

TASK PARALLELIZATION The edges of the task graph are later translated directly into OpenCL dependencies during the replay stage. Hence it is crucial that these dependencies are minimal to expose a maximum amount of task parallelism. This can be measured by the width of the graph: the optimizer aims to increase the width and decrease the length of the critical path. The delay mechanism avoids conflicts between actions in a very conservative way by preventing them to use conflicting objects at the same time. However, a producer/consumer approach, allowing multiple actions to share read-only objects is a much more precise dependency management and also provides the same guarantees. Each action is classified as a consumer, which only reads data, or a producer, which creates or modifies data. Doing this requires more information about the kernels than is available from the runtime alone. Hence, HELIUM combines the runtime information with static code analysis.

KERNEL-AWARE DEPENDENCY ANALYSIS Much more contextual information is known about each kernel instance during the optimization stage compared to ahead-of-time compilation. As seen in the analyzer, the command functor knows the iteration domain definition and all the kernel parameters.

This information is combined with the *PADs* generated during the static analysis of the example shown in Figure 4.15. In the example, the *PADs* for kernels A and B and their parameters are represented as:

$$PAD(A) = \forall gid_0 \in [0..100[\begin{cases} LD_{glob}(a + sizeof_{int} * gid_0), \text{ with } a = \mathbf{b1} \\ ST_{glob}(b + sizeof_{int} * gid_0), \text{ with } b = \mathbf{b2} \end{cases}$$

$$PAD(B) = \forall gid_0 \in [1..100[\begin{cases} LD_{glob}(t + sizeof_{int} * (gid_0 - 1)), \text{ with } t = \mathbf{b1} \\ ST_{glob}(u + sizeof_{int} * gid_0), \text{ with } u = \mathbf{b3} \end{cases}$$

From this, the optimizer deduces that b1 is only read from both kernels since it only appears in *LD PADs*. As the *ST PAD* sets are disjoint for both kernels, there is no data dependency between them and they can execute in parallel. The optimizer deletes the edge to disconnect the two commands, as shown in Figure 4.16a.

In addition to performing comparison between read and write sets, the optimizer also performs a range-based comparison over the iteration domain to allow multiple writers to modify the same data in parallel if they operate on disjoint data ranges. This does not necessarily mean that the iteration domains must be disjoint since the domain does not necessarily correlate to the memory accesses. Instead HELIUM uses symbolic execution of the *PADs* to detect eventual conflicts.

TASK-AWARE DEPENDENCY ANALYSIS After each memory object that is manipulated by the kernel instances has been marked with read-only or read-write access, this information is used to eliminate dependencies with non-compute nodes. Memory operations can easily be classified as consumers, like a read or a copy source, or a producer, like a write or a copy destination. The same technique is used to delete the edges between multiple readers. However, a writer must acquire the state of the object to perform the operation: dependencies are created between a writer and all the precedent readers to avoid data

races. In the example the buffer b2 is produced by the invocation of A, and subsequently consumed by the read operation and an instance of C. Since they are both readers, the dependency between them is deleted. They are both connected to the last producer of b2, which was the instance of A. This allows parallel execution of the memory operations and the compute operations.

TASK POSTPONEMENT While the normal execution of common synchronizations primitive like *clFinish* or a blocking operation on an in-order queue would force the execution of all pending commands, HELIUM lazily evaluates individual commands. If a command is not a dependency of a command having a side effect on the host application, it will not be executed before the synchronization point. This effectively postpones all kernel instances as much as possible, until they have to be evaluated, which reduces the synchronization delays and allows for more optimizations by aggregating more tasks. In the example, the second instance of A produces b2, which is not a dependency of any memory operation. Therefore, it can be delayed until after the synchronization, as shown in Figure 4.16c.

EXPLICIT DEPENDENCY ELIMINATION Finally, the dependency analyzer refines the user provided dependencies, if there are any. The delay mechanism already removes the implicit dependencies of in-order queues. However, the host application might be using multiple queues, or an out-of-order queue, in which case an event-based synchronization must be used. Since the analyzer in HELIUM guarantees that data races will be avoided even across queues, the user-specified dependencies can be deleted most of the time and replaced with a minimal set computed by the optimizer. They are only used to solve rare conflicts, like multiple unsequenced writers.

4.4.2.2 *Task Optimizations*

The task optimizer reduces the number of nodes, by combining or deleting them. This provides a high level abstraction over the compiler transformations: instead of generating code incrementally, HELIUM uses properties of the nodes and performs lightweight transformations on the task graph. This produces the final schedule shown in Figure 4.17d.

HELIUM employs a greedy optimization strategy where all the transformations are considered for each action as soon as it is delayed, in the order listed in this section. To guide

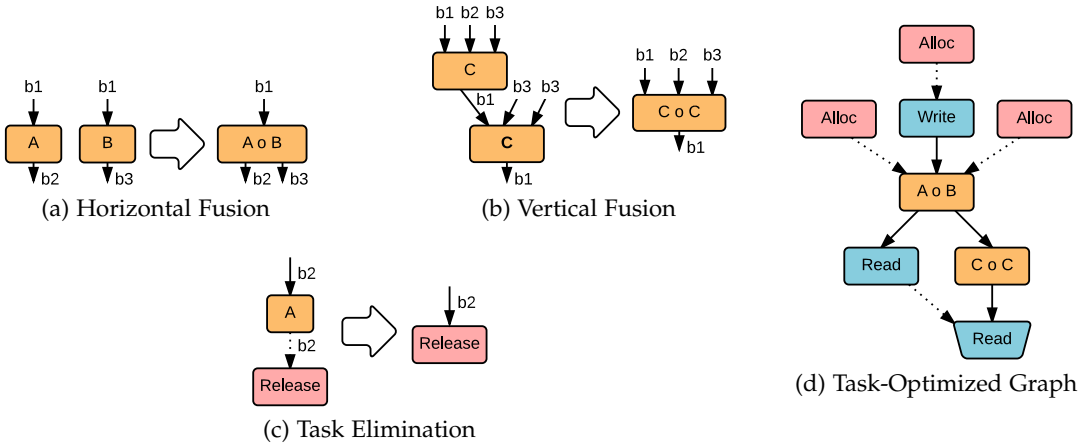


Figure 4.17: Task Graph Node Optimizations: Helium uses the optimized dependencies and the static analysis of the device code to merge data independent nodes (a) or data-dependent kernels (b) whenever it is possible and beneficial. Tasks are also removed from the graph whenever the host application cannot use their outputs (c).

the optimization process, simple heuristics are put in place to estimate the benefits of a transformation. These heuristics are listed alongside the optimization description.

DATA INDEPENDENT FUSION The task graph generated by the delay phase and optimized by the dependency analyzer provides the guarantee by construction that there is no data race between any two unconnected nodes. Independent nodes can therefore be executed in any order, or even evaluated at the same time, by *fusing* multiple kernel instances into one.

However, some restrictions of the OpenCL model limit the applicability of horizontal fusion. Since every thread in a workgroup can be synchronized using a barrier, the semantics of this barrier, and hence the size of the workgroup, must be preserved. Two kernels can still be fusable even with different workgroup sizes if at least one of them does not use any barrier. In this case Helium will override the provided workgroup size and generate functions to compute the local size and index within the fused kernels.

While it would be safe to fuse any two fusable nodes in the graph, Helium uses the *PADs* built during device code analysis to estimate the performance benefits of merging the kernels.

In order to identify good fusion candidates, the memory accesses are compared across kernel instances. A small constant difference between two *PADs* expressions using the same

base pointer indicates neighboring accesses which can be optimized. In this case, both instances access an address of $sizeof_{int}$ bytes apart within each thread:

$$(\mathbf{b1} + sizeof_{int} * gid_0) - (\mathbf{b1} + sizeof_{int} * (gid_0 - 1)) = -sizeof_{int}$$

Since this is small enough for most hardware to optimize against, the two nodes in the graph are fused and their edges merged, as shown in Figure 4.17a.

VERTICAL FUSION An edge in the graph always represents a producer/consumer dependency between two actions. However, the second task does not always have to wait for the first task to complete entirely before starting: streaming data from one task to the other can be implemented at a more fine-grained level. The OpenCL model does not provide synchronization between any two threads, so this transformation is only applicable under very strict circumstances: there must be no concurrent memory accesses to the same address within or across the kernel instances at a thread level.

Concurrent accesses within each instance are fairly rare since reading and writing different positions of the same buffer can easily lead to race conditions if not carefully implemented. The absence of dependency can be verified by comparing load and store *PADs* in each instance. In the example, the execution context can be injected into the *PADs* for the two instances of C, note C_1 and C_2:

$$PAD(C_1) = \forall gid_0 \in [2..98[\begin{cases} LD_{glob}(\mathbf{b1} + sizeof_{int} * (gid_0 + 2)) \\ LD_{glob}(\mathbf{b2} + sizeof_{int} * (gid_0 + 2)) \\ LD_{glob}(\mathbf{b3} + sizeof_{int} * (gid_0 - 2)) \\ ST_{glob}(\mathbf{b1} + sizeof_{int} * (gid_0 + 2)) \end{cases}$$

$$PAD(C_2) = \forall gid_0 \in [0..100[\begin{cases} LD_{glob}(\mathbf{b1} + sizeof_{int} * gid_0) \\ LD_{glob}(\mathbf{b3} + sizeof_{int} * gid_0) \\ ST_{glob}(\mathbf{b1} + sizeof_{int} * gid_0) \end{cases}$$

The only load and store *PAD* using the same base pointers is for b1 in both kernels, and they are to the same address within both instances, so there is no spatial dependency. The stores from the producer and the loads from the consumer are compared across kernels to check for dependency. In this case, there is a conflict:

$$\begin{cases} ST_{\text{glob}}(\mathbf{b1} + \text{sizeof}_{\text{int}} * (\text{gid}_0 + 2)) & \text{, with } \text{gid}_0 \in [2..98[\\ LD_{\text{glob}}(\mathbf{b1} + \text{sizeof}_{\text{int}} * \text{gid}_0) & \text{, with } \text{gid}_0 \in [0..100[\end{cases}$$

This introduces a spatial dependency: for example, the first thread of C_1 executes a store operation at the fourth element of the array, which is loaded by the fourth thread of C_2. Since the threads would execute in parallel when fusing, this would introduce a race. However, the applicability of this optimization can be extended by performing further analysis.

Using a technique similar to loop alignment [FHM99], the iteration domain can be translated and compensated by a constant in the *PADs*. A translation factor in introduced in the producer domain to align conflicting loads and try to solve the system of (one) equation in this case:

$$\{\mathbf{b1} + \text{sizeof}_{\text{int}} * (\text{gid}_0 + \mathbf{2}) = \mathbf{b1} + \text{sizeof}_{\text{int}} * (\text{gid}_0 + \mathit{trans})\} \Rightarrow \{\mathit{trans} = 2\}$$

Since the system has a solution, there exist a translation which solves the spatial dependency. The translation is applied to the domain of C_1 and compensated in its *PAD*:

$$PAD(C_1 \circ C_2) = \forall \text{gid}_0 \in [0..100[\begin{cases} LD_{\text{glob}}(\mathbf{b1} + \text{sizeof}_{\text{int}} * \text{gid}_0) \\ LD_{\text{glob}}(\mathbf{b3} + \text{sizeof}_{\text{int}} * \text{gid}_0) \\ LD_{\text{glob}}(\mathbf{b2} + \text{sizeof}_{\text{int}} * (\text{gid}_0 - 4)), \text{ if } \text{gid}_0 \geqslant 4 \\ LD_{\text{glob}}(\mathbf{b3} + \text{sizeof}_{\text{int}} * (\text{gid}_0 - 4)), \text{ if } \text{gid}_0 \geqslant 4 \\ ST_{\text{glob}}(\mathbf{b1} + \text{sizeof}_{\text{int}} * \text{gid}_0) \end{cases}$$

After the translation removed the dependency, the two kernels can now be fused since there is no data race between individual threads. The transformation is represented in

Figure 4.17b. The *PADs* of the fused kernel $C_1 \circ C_2$ contains one fewer load and one fewer store than the sequence $C_1 \mapsto C_2$.

TASK ELIMINATION Executing only the tasks which have side effects on the host application accumulates all the tasks that do not. Some of these tasks, however, might never have any side effect; these are called *dead tasks*. This artifact occurs when the output of a task is overwritten by a subsequent task, or when it is no longer accessible from the host application and not used by any other task. Dead tasks are difficult to spot in the code since they might result from highly dynamic and complex flow, or emerge as an unwanted consequence of the optimization process.

Dead tasks are never executed in HELIUM, since all tasks are lazily instantiated. However, they have undesirable side effects, like preventing de-allocation of the objects they are associated with, or clobbering the task graph. Hence they have to be eliminated.

HELIUM spots when a task becomes dead by tracking the use and lifetime of each memory object. When the reference count on a buffer becomes zero, or when a buffer is overridden by a subsequent action, the edge from the last producer of this object in the task graph is marked as defunct. When all outgoing edges of an action are defunct, it can no longer have any side effect and can safely be evicted from the task graph.

In the example, b2 is released after the synchronization point and was last produced by an instance of A enqueued before the synchronization point. Since b2 is the only output of this instance, it can be eliminated and is removed from the task graph, as shown in Figure 4.17c.

4.4.2.3 *Final Optimized Graph*

OPTIMIZATION SUMMARY This subsection presented the task graph optimizer and the transformations applied to refine the dependencies and reduce the number of commands. Although the optimizer makes use of the low-level compiler analysis to validate the legitimacy of the optimizations, the transformations themselves are applied using a very high level representation. All transformations consist only in manipulating nodes and edges. This is a lightweight process which relies on the sound abstraction model to introduce concepts which are hard to implement correctly by hand, like task parallelism and data-race avoidance.

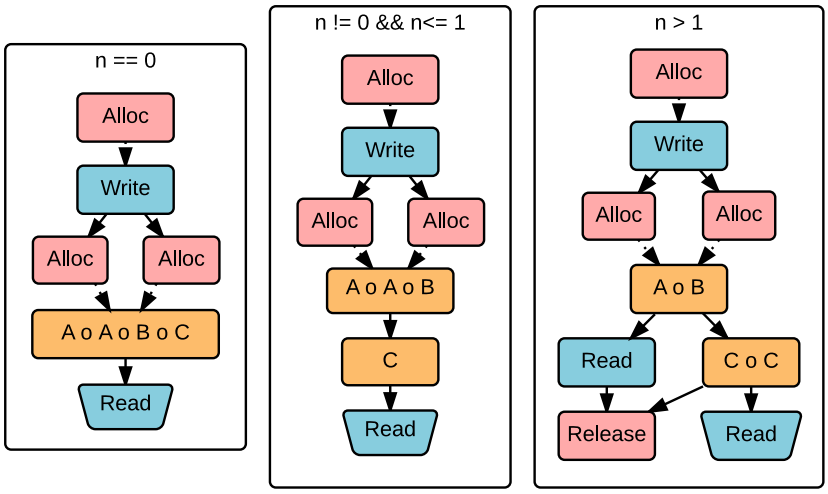


Figure 4.18: Optimized task graphs depending on the runtime value n. Implementing all the variations by hand would require a complex control flow in the host optimization and duplications in the device code.

CONTEXT SENSITIVITY Combining these optimizations incrementally creates a highly runtime specific task graph which takes into account the actual dataflow path. Figure 4.18 shows the three possible optimization sets depending on the value of n in the example application. If it had been optimized manually, the application would have required four additional specialized kernels: $A \circ B$, $C \circ C$, $A \circ B \circ C$ and $A \circ A \circ B \circ C$, with high code redundancy. This would have also considerably increased the complexity of the host application since each dataflow path must be implemented in distinct control flow path, which may extend beyond this code fragment to the rest of the application.

Once the task graph has been optimized, it is executed by the OpenCL vendor implementation during the replay phase.

4.4.3 *Replay Mechanism*

The delay mechanism is in charge of restoring the coherency between the host and device programs. After a synchronization point is reached and the task graph has been optimized, it has to be translated into OpenCL function calls and executed by the vendor implementation. Before executing the commands, the high level transformations need to be materialized and the code is further optimized. The commands are then efficiently dispatched.

4.4.3.1 *Materialization of the Optimizations*

Before the task graph is executed, the transformations applied by the optimizer have to be generated. This involves generating and compiling new kernels on the fly. HELIUM's compiler is based on LLVM and generates either SPIR or PTX binaries, which are loaded back into the vendor implementation. Since compilation is relatively expensive, HELIUM only generates code after the optimization stage, and uses the same lazy evaluation technique to materialize the transformation only for the nodes which have to be immediately replayed. The JIT optimizations and the code staging mechanism are described in this subsection.

CONSTANT PROPAGATION At this stage, variables such as the kernel arguments or the global and local domain sizes are known and can be used by the compiler to simplify the code. However, HELIUM tries not to over-specialize the code by only applying constant propagation if it yields good potential for optimizations. Only scalar values controlling loop trips and branches are propagated, since these are the most aggressive optimizations.

HELIUM uses the type information contained in the IR to emulate reflection and undo the type erasure introduced by binary argument storage in OpenCL. This allows the optimizer to decompose any user-defined type into primitive types and fetch individual members if necessary to propagate as few variables as possible.

ALIAS ANALYSIS In the same way that constant propagation can retrieve values of primitive types from the argument list, the optimizer can extract pointer arguments and compare their values. If multiple arguments match the same buffer handle within a kernel instance, the pointers will alias on the device.

The compiler explicitly replaces all uses of aliased pointers with a unique argument, and applies redundant and dead load elimination passes to reduce the amount of memory requests. This transformation must respect memory fences: it assumes any unknown function manipulating a pointer might clobber memory and identifies direct calls to *mem_fence*. Memory accesses are not reordered or eliminated across these points.

KERNEL FUSION The "fused" nodes from the task graphs simply contain the subgraph of all the fused kernel instances and their dependencies. This is used by the compiler to generate specialized functions by inlining the calls.

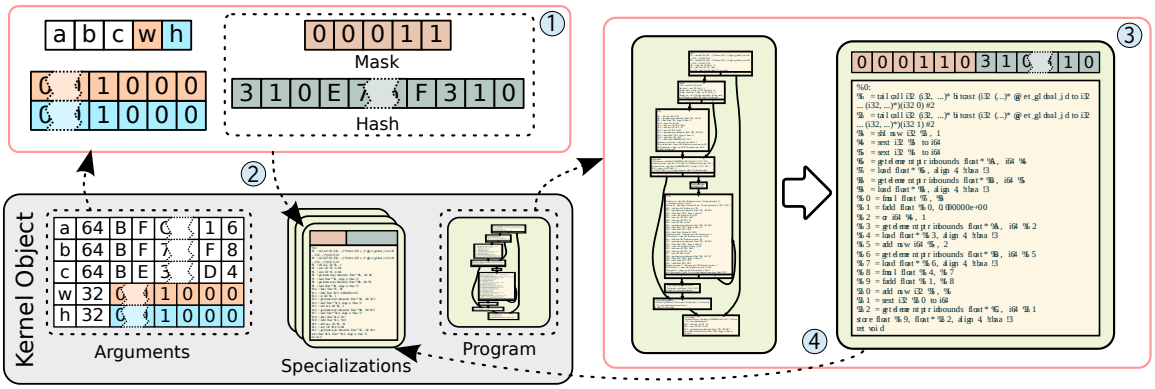


Figure 4.19: Specialization process for constant propagation: a specialization key is generated from the argument (①). If a specialization lookup (②) does not find an existing specialized version of the code, it is compiled (③) and registered (④).

The same mechanism is used for both horizontal and vertical fusion: a new kernel function from aggregating the arguments and inlining the bodies of each instance into a new function. HELIUM does not attempt to re-use local memory across fused instances: the amount of local memory used by the fused kernel will be equal to the sum of the local allocations in the fused instances.

The kenels can be inlined in any order for horizontal fusion while vertical fusion must preserve the execution ordering and inlines them in topological order. The compiler then generates the necessary code for the kernels to execute correctly: if the domains are not identical, compiler guards are generated around the body of each kernel to prevent execution outside the original iteration space. Additional code is generated to compensate for domain translation introduced by the vertical fusion and to re-adjust the local workgroup size and index if it has been overridden.

CODE STAGING In order to reuse the specialized code multiple times during an execution, the output of each transformation is stored in a compile cache. To improve the chance of reuse, the compiler generates generic code to avoid overspecialization.

In order to find whether a specialization already exists in the compile case, the kernels are fused in a predictable way to allow a lookup by name. Kernels are inlined in alphabetical order whenever possible, and the name of the fused kernel is an aggregation of the original kernel names. A unique identifier is generated from propagating constants and pointers to guarantee that the specialization is an exact match. Figure 4.19 illustrates the process for scalar expansion of two arguments in the original matrix-matrix multiply exam-

ple shown in Figure 4.3. The first three arguments are distinct pointers and are not used for optimization. The last two arguments, w and h are used to control the nested loops: they are good candidates for specialization. The compiler creates a mask to flag which values are to be propagated, and a unique hash is generated from their aggregation (step ①). Both the mask and the hash are used to do a lookup in the specialization cache (step ②). If an optimized version already exists with the same attributes, it can be used directly. Otherwise, the JIT compiler is invoked to create a new version of the kernel (step ③) which is then registered using the mask and hash (step ④). Subsequent calls using the same specialization identifier can use the pre-compiled version of the code.

4.4.3.2 *Scheduling and dispatch*

Translating the final optimized task graph into OpenCL functions and invoking the vendor implementation is the last step necessary to restore coherency at a synchronization point. This process is straight forward since the graph itself contains all the information needed in terms of command context and dependencies. We briefly describe the dispatch mechanism and conclude this section by comparing the original and optimized executions.

COMMAND DISPATCH At the start of the program, HELIUM creates three out-of-order queues for each device in order to maximize the chance of task parallelism: one is used for the kernel invocations and the other two for memory operations, read and write respectively. The scheduler traverses the list of pending tasks: the dependencies of each read operation are pushed in the corresponding queue in topological order. Each command function is able to generate the original function call with the appropriate parameters. This might transparently trigger additional calls, like restoring the state of the arguments for a kernel before invoking it. A *cl_event* is attached to each command in the queue: it is used to translate the edges in the graph. The events of all the connected nodes are first gathered before the command is pushed in the queue with these as dependencies. This exposes the maximum amount of task parallelism since the queues are out-of-order[2] and the dependencies are optimized. Finally, the scheduler waits for all three queues to complete all actions by calling *clFinish*. The coherency between host and device is now fully restored and the ex-

2 HELIUM still uses three separate queues since some vendors have been experimentally shown to serialize communication when using a single out-of-order queue.

```
cl::Event e1, e2, e3, e4, e5;
std::string c = R"(
 #define ID get_global_id(0)
 #define buf global int*
 kernel void AoB(buf at, buf b, buf u)
   { b[ID] = 2 * at[ID];
     if(ID>0) {u[ID] = at[ID-1] - 1;} }
 kernel void CoC(buf x, buf y, buf z)
   { if(ID >= 4) y[ID] += x[ID-4] + z[ID-4];
     y[ID] += 2 * z[i]; } )";
Program p{cx,{1{c.data(),c.size()}}}; p.build(devs);

Buffer b1{cx,f,siz};
qo.enqueueWriteBuffer(b1,false,0,siz,data,NULL,&e1);

Buffer b2{cx,f,siz}, b3{cx,f,siz};
Kernel AoB{p,"AoB"}; AoB.setArgs(b1,b2,b3);
qc.enqueueNDRangeKernel(AoB,{0},{100},{1},{e1},&e2);

qi.enqueueReadBuffer(b2,false,0,siz,tmp,{e2}, &e3);

Kernel CoC{p,"CoC"}; CoC.setArgs(b2,b1,b3);
qc.enqueueNDRangeKernel(CoC,{0},{100},{1},{e2}, &e4);

qi.enqueueReadBuffer(b1,true,0,siz,reis,{e3,e4});
```

Figure 4.20: OpenCL representation of the code form Figure 4.15 as executed by the vendor implementation. HELIUM transparently reduced the number of commands, exposed task parallelism though fine grained event synchronization and generated a specialized version of the device code.

ecution is semantically identical to the original command sequence. The cycle then restarts until the next synchronization point is reached.

THE REPLAYED PROGRAM The result of the replay phase is transparent to the target application but is able to perform very aggressive transformations. Figure 4.20 represents an equivalent OpenCL program to Figure 4.15 as seen and executed by the vendor implementation. The optimized code contains fewer OpenCL commands and uses fine grained synchronization. The device code has been entirely re-written in this instance since the original kernels are not needed for execution but replaced with specialized versions. Coding these optimizations by hand would require explicitly implementing all the possible control flow in separate branches along with the exact preconditions necessary for each optimized flow. This would lead to a high amount of code redundancy and very poor maintainability.

4.5 LIMITATIONS

While the delay-replay mechanism is entirely transparent and the correctness of the computation is guaranteed by avoiding data races, Helium can nonetheless have undesirable side effects which can negatively impact the host application. This section briefly lists the limitations of the delay-replay model, which cannot be solved by extending the framework.

ERROR HANDLING Most OpenCL commands either return an error code or take a pointer on an error flag as a parameter in order to check whether the command executed successfully from the host program. However, since the delayed actions are not executed by the vendor implementation, it is not always possible to detect an error. Helium emulates some of the error checking – like verifying that the handles are valid – but it cannot spot runtime errors. In this case, the error code is forced to the success value when the command is delayed. During the replay phase, the error code is always automatically checked and the program will be aborted on error. However, this may impair debugging as tracking the origin of the error might be difficult.

APPLICATION PROFILING It is possible to get profiling information from some commands using event objects. Many applications use this as a mean of benchmarking compute performance of the accelerator. However, there might be a mismatch between delayed and replayed events. In some cases, the framework can automatically detect this problem and avoid it: for example, getting profiling information about a task which has been eliminated returns zero in all fields. While it might look inconsistent from the host application, it allows profiling calculations to be accurate. Another problem is caused by kernel fusion: the host program has several event handles on multiple kernel instances, but they all correspond to a unique real invocation. In this case, one of the events will hold the profiling information, and the others will be initialized to zero. This might cause issues if the host compares the values to detect an eventual overlap for example. The last timing issue arises if the application is profiled in distinct stages separated by a synchronization operation. Since the optimizer can move individual commands across synchronization points, the time measured might not be representative of the list of actions enqueued between them.

HOST DEVICE OVERLAP Delaying the OpenCL commands might cause some overheads since it prevents host and device from performing computation in parallel. However, in

practice most applications delegating computation to an accelerator actively wait for the result shortly after issuing a chain of commands in order to proceed further, and so this does not cause significant delays.

These limitations do not affect the correctness of the application, and in most cases they do not affect performance measurement either. However, these slight behavioral changes prevent Helium from being implemented as a default behavior for a vendor implementation, since they might impair programmability during the development phase.

Note that the transformations have not been formally proven to be correct, as the current implementation is a prototype of delay-optimize-replay scheduling. Implementing this strategy as part of the OpenCL model would require this verification stage.

4.6 EVALUATION METHODOLOGY

4.6.1 *Experimental Setup*

The machine used for the test has an Intel Core i7-4770K CPU with 16 GB of RAM and an Nvidia GeForce GTX 780 GPU connected via PCI-E 3.0. The OpenCL 1.1 implementation is provided by Nvidia's Linux driver 331.79. Since Helium's backend relies on Nvidia's open source PTX backend, only the performance on the GPU can be measured.

4.6.2 *Evaluation Methodology*

Helium is evaluated on a collection of applications available in both raw (baseline) and hand-optimized forms. The baseline version prioritizes code simplicity and modularity, while the hand-optimized versions apply the transformation described in Section 4.2 iteratively as much as possible on the static parts of the applications. This enables us to evaluate the benefits Helium brings directly *against* hand-optimization and the benefits it brings when applied *after* hand-optimization. As explained in Section 4.2, opportunities for hand-optimization of dynamic applications are quite restricted because of the difficulty of verifying the validity of the transformations, and the latter comparison demonstrates that they may even be counter-productive when a powerful, automated system such as Helium is available. The four benchmarks in the test suite are:

- *CCO*: a simplified version of the Copy-Compute Overlap benchmark from the Nvidia benchmark suite. Two buffers are written to the device, read by a compute kernel which stores its output in a third buffer, and the result is copied back. This compute sequence is repeated for a given number of iterations. The optimized implementation fragments the computation in by splitting the input and output buffers in half. The commands are then pushed in two in-order queues in a very specific order such that computation and communication may overlap across iterations. The baseline implementation is a simplified version of the code where all actions are pushed in a single in-order queue.

- *Sobel*: The Sobel filter is a discrete differentiation operator commonly used for image processing applications like edge detection. In its general form, two gradient convolution operators are applied to the input, generating two temporary values, which are combined by a third operation. In this case, the code can be hand-optimized but in more complex applications these operators are created by composition of simple filters, resulting in a number of combinations that are impossible to optimize by hand.

- *geoMatrix*: computes the entrywise geometric mean of n matrices by computing n-1 Hadamard products and a pointwise division. This process is present in many signal and image processing applications, such as lossy image compression algorithms. The baseline implementation composes the operations using generic binary functions. The hand-optimized version uses specialized kernels for processing multiple matrices in-place and defines a tree reduction for the multiplication stage.

- *ExpFusion*: This application fuses several images taken with different exposure times into one to increase the dynamic range by using a Laplacian decomposition and a Gaussian pyramid [MKR07]. The depth of the pyramids, number of input images and their properties are not known statically, making the application highly dynamic. The hand-optimized version makes some assumptions about the input to fuse some static parts of the pipeline (which can only apply to a subset of possible inputs).

Each application is tested with two different input sizes. The large input size is 4 times larger than the small one for all input and output buffers. *CCO* is tested with input arrays of 20K and 80K elements. *Sobel* and *ExpFusion* use input images of $2K \times 2K$ and $4K \times 4K$ pixels. *geoMatrix* uses matrices of $4K \times 4K$ and $8K \times 8K$ elements. For *geoMatrix* and *ExpFusion*, we

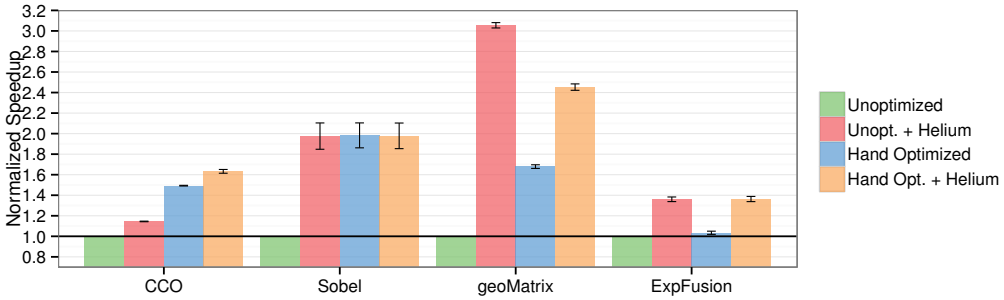


Figure 4.21: HELIUM performance for to non- and hand-optimized code. We report the speedup over the unoptimized version across input sizes and its 99% confidence intervals with and without HELIUM.

also increase the number of inputs: *geoMatrix* is tested with 16 and 32 input matrices; *ExpFusion* with 4 and 12 input images and a pyramid depth of 4 and 8 respectively.

Each benchmark is executed ten times with and without pre-loading our framework, measuring the total wall clock time on the host between the first OpenCL action pushed in a command queue to the termination of the last synchronization or blocking operation. We report the analysis for the interquartile mean across sixteen runs. As both the Nvidia driver and HELIUM use a persistent compiler cache, most of the compilation overhead is excluded from the measurements. The overhead of the analysis and task graph transformation, however, are still present since the trace is regenerated with each execution.

4.7 EXPERIMENTAL RESULTS

Figure 4.21 summarizes the effect of HELIUM on the four applications across different problem sizes. HELIUM improves performance over the unoptimized version in all cases, and over hand-optimized version in all but one case – where manually changing implementation strategy outperforms HELIUM. It never has a harmful effect.

Table 4.2 summarizes the experimental results in more detail. For each application and input, we report the number of OpenCL commands pushed in the command queue for the baseline application and how many of those were kernel invocations. We then compare to three alternative executions: the same baseline binary with HELIUM preloaded, a manually hand-optimized version and lastly HELIUM with the hand-optimized binary. For each alternative execution we report the performance relative to the baseline application and the number of commands actually executed by the vendor implementation. The findings for each application are discussed below.

	Baseline	Baseline + Helium		Hand Opt		Hand Op + Helium	
	# Tasks Enqueued	# Tasks Replayed	Speedup	# Tasks Enqueued	Speedup	# Tasks Replayed	Speedup
CCO			**1.14**		**1.49**		**1.63**
small	40 (10)	40 (10)	1.14	80 (20)	1.50	80 (20)	1.63
large	40 (10)	40 (10)	1.15	80 (20)	1.49	80 (20)	1.63
Sobel			**1.98**		**1.98**		**1.98**
small	5 (3)	3 (1)	1.78	3 (1)	1.80	3 (1)	1.79
large	5 (3)	3 (1)	2.17	3 (1)	2.17	3 (1)	2.17
geoMatrix			**3.06**		**1.68**		**2.45**
small	19 (18)	2 (1)	3.05	10 (9)	1.68	2 (1)	2.47
large	35 (34)	2 (1)	3.06	18 (17)	1.68	2 (1)	2.44
ExpFusion			**1.36**		**1.04**		**1.36**
small	446 (441)	298 (293)	1.32	339 (334)	1.05	298 (293)	1.31
large	2198 (2185)	820 (807)	1.41	1727 (1714)	1.02	820 (807)	1.41

Table 4.2: Performance impact of HELIUM on non-optimized baseline and hand-optimized version. For each application we report how many commands are issued, and how many of those are kernel invocations in parenthesis. All speedups are relative to the non-optimized code and include profiling and compilation overheads.

CCO HELIUM is able to introduce task parallelism from the baseline using its parallelizing scheduler. While the device only has one DMA engine, the out-of-order queues reduce the delays between commands, resulting in a speedup of 1.15x. The hand-optimized version does a better job by fragmenting the computation in smaller units, increasing the scope for copy-compute overlap, achieving 1.49x. However, when combining HELIUM and the hand-optimized version, we get the best performance at 1.63x. HELIUM also takes advantage of the fragmented tasks but dispatches the tasks more efficiently. Note that if the device had two DMA engines, HELIUM would automatically exploit duplex communication for both the hand-optimized and baseline versions. Doing so manually would require a complete re-write of the already hand-optimized code.

This shows that while HELIUM is able to improve the performance of existing code, it may also benefit from hand-transformations exposing more optimization opportunities.

SOBEL The first two stages being data independent and their output being processed by a map function, the three kernels can safely be merged into one, resulting in two fewer store instructions per point (for each temporary buffer) and three fewer load per point (two for the temporary and one redundant read). When isolating this pattern in a single application, the same conclusion can easily be reached by a programmer, who applied the same optimization strategies in the hand-optimized version. All implementations generated the

same code and achieve the same speedup of 1.98x. However, if this pattern is part of a larger image application, or if it the result of dynamically composing from operators at runtime, it would become increasingly difficult to optimize by hand.

This application shows that HELIUM performs the same transformations as an expert programmer, without bloating the code base with specialized versions of the code.

GEOMATRIX Because the number of input matrices is not known statically, it is not possible to implement a specialized version of the kernel. The hand-optimized implementation achieves a speedup of 1.68x using in-place operations and creating specialized operators to multiply three matrices at once. HELIUM also generated specialized kernels, but combining all the matrices at once since their number is known at runtime. For both input sizes, HELIUM generated a single kernel, improving performance by over 3x.

While opting for a reasonable strategy of parallelizing tasks in the multiply stage, the manual transformations did not result in an important performance improvement. Task parallelism is not exploited by the GPU in this case because both inputs generate enough threads to occupy the entire device. However, this optimization considerably increases the complexity of the dataflow paths, resulting in poorer performance gains by HELIUM compared to HELIUM operating on the baseline. In both cases, our system performs the same optimizations: all kernels are fused into a single specialized kernel, and the amount of computation is roughly the same in both cases. The difference comes from the use of memory. By optimizing a composition in the baseline, the optimized version resulted in a single load per input matrix element and a single write for the result. The hand-optimized version uses writes to temporary buffers to speed up the reduction tree stage. These writes cannot be eliminated, as they are not released until after reading the final result. Hence, their side effect cannot be predicted and HELIUM cannot eliminate the dead store operations.

This demonstrates that partial hand-optimization can actually be counter-productive if it is attempted prior to automatic techniques.

EXPFUSION Exposure fusion is a highly dynamic application and very little can be known statically. First, the pre-processing step depends on the image format, which is unknown at compile time. Second the number of images is unknown. A Laplacian Pyramid of an arbitrary depth is then built by recursively blurring and down-sampling the images,

and the final image is reconstructed using a Gaussian pyramid. Despite the complexity of the pipeline, the host program only requires less than a hundred lines of code, but it contains complex control flow and requires a very modular design code to be maintainable.

The number of kernels executed varies depending on the input parameters, making manual optimizations very difficult, even with the help of a profiler. The small input size used 4 RGB input images and a pyramid depth of 4, which generated over 400 kernel instances. The larger input used 12 images and a depth of 8, invoking more than 2000 kernels.

The hand-optimized version cloned and specialized a large part of the code for three-channels images, leaving the original generic application as a fallback if this assumption is not met at runtime. By specializing (hence duplicating) kernels, the overall number of invocations decreased by 20%. However, the performance gain was less than 5%, since most of the time is spent in combining across images rather than across image channels.

Helium can take advantage of runtime specialization to achieve the same speedup of 1.36x from either the baseline or the hand-optimized code. It generates specialized code which combines all the images at once, leaving only the convolution steps in between, which cannot be merged due to spatial and temporal dependencies.

As the pyramid gets wider and deeper with the larger input size, Helium shows better scalability than hand-optimized code, achieving a speedup of 1.4x while the hand-optimized version did not improve. This shows that even with more versions of the specialized kernels in the hand-optimized implementation, Helium will always stay ahead by generating them at runtime, allowing it to adapt to new inputs.

This last application demonstrates the applicability of Helium on large and dynamic workloads where hand-optimizations are not applicable or not efficient.

4.8 SUMMARY

This section has presented Helium, a transparent OpenCL overlay for automatically computing and optimizing task graphs in OpenCL applications. It is based on a delay-optimize-replay mechanism, allowing the scheduler to chain OpenCL commands together according to their dependencies and compute only what is necessary in the host application. The optimizer also performs other types of transformations on the task graph, such as task re-

ordering and kernel fusion, which improve the overall performance by increasing device occupancy and simplifying memory transactions across kernels.

We evaluated this framework on multiple benchmarks to assess both the efficiency of Helium's compiler transformations and its parallelizing scheduler. We found that in most cases Helium can replicate the performance of hand-optimized code without the expense of refactoring. For highly dynamic applications Helium can outperform hand-optimized code by taking advantage of runtime information. Finally, we showed that over-engineered and over-complicated code not only impairs maintainability but may also harm automated optimization processes, which could achieve better results with simpler code.

This method demonstrates the strength of lazy evaluation as an alternative execution model for OpenCL command queues. The existing standard concepts of JIT compilation and tasking model are combined as a powerful iterative compilation and staging framework without any modification of the source code.

5 AUTO-TUNING MULTI-GPU STENCIL COMPUTATION

In the previous chapter, we showed that applications can be automatically optimized within the OpenCL model to enhance productivity and performance. However, properties of the underlying hardware exhibit behaviors and artifacts which cannot be expressed directly in this model; and yet have a great impact on performance. Programmers usually bridge this gap by implementing ad-hoc optimizations for a particular system, baking hardware knowledge into their optimization strategy. This process considerably degrades productivity, and the resulting applications are not performance portable.

This chapter investigates automated hardware specific optimizations and how they can be related to the OpenCL model. More specifically, we analyze the impact of interconnect mediums on data communication in a multi-device context. The growing number of devices in a single system and their increasing compute power are saturating the data bandwidth on shared communication channels and limiting scalability. We demonstrate this by exploring a particular application domain: stencil computations. We implement a high level framework for stencil computation with an embedded auto-tuner to optimize data placement and communication patterns. An exploration of the space reveals that PCIe buses, despite their uniform appearance, exhibit heterogeneous characteristics. While this information is not accessible programmatically, our auto-tuner can discover it and exploit it to improve performance by up to 23%.

This chapter is organized as follows: Section 5.1 motivates the importance of multi-device optimization and how this affects stencil computations. Section 5.2 presents some optimization strategies for distributing stencil computations and their parameters. Section 5.3 describes Partans, a high-performance and high-productivity distributed stencil computation framework. Section 5.4 introduces our experimental setup and Section 5.5 discusses the results of our evaluation.

5.1 MOTIVATION

Heterogeneous systems are getting increasingly complex; not only in the microarchitecture making up each device but also as entire systems. The number of devices, like the number of cores on them, is increasing. Most commodity computers provide up to four PCI slots hosting accelerators; and more specialized hardware like the TYAN FT72B7015 or the Dell PowerEdge C410x can provide up to 8 and 16 slots respectively. Graphics cards like the AMD 5970 or the Nvidia GTX 590 integrate multiple GPUs on the same PCIe slot. Hence, complex systems can be built with a large number of devices, and a tremendous theoretical compute power. For such systems, relying solely on OpenCL will simply not scale well, since other parts of the system which are not expressed in the model will hit hardware limitations. Thus, even approaching peak performance in a single application remains elusive in practice.

Knowledge about the underlying system allows the implementation of hardware specific optimizations, which utilize the resources in a more efficient way. However, these optimizations are applied manually on an ad-hoc basis, after a thorough investigation of the bottlenecks in a given system. Hence they require a lot of effort, and the result is not performance-portable. While the OpenCL model alone does not provide enough information to implement hardware specific optimizations in a non-intrusive way like Helium, this chapter establishes that abstraction layers can nonetheless be implemented with hardware sensitive optimization strategies. We demonstrate this technique on a particular domain: stencil computations.

The stencil computation pattern is fundamental in many domains. Its characteristics have been introduced in Section 2.6 and some of the extensive research on optimizing this particular skeleton was presented in Section 3.3. Since stencil computations are embarrassingly parallel, memory intensive and very regular, they have been shown to be a good match for GPUs, which offer a powerful and energy-efficient solution for these problems.

However, many stencil applications are trying to solve problem sizes that could not fit on any existing GPU. Complex quantum physics simulations, for example, require four-dimensional structures with a large amount of data at each point in the domain. A single GPU in this case cannot simulate more than a handful of particles at a time, which is a severe limitation. This class of applications generally aims towards weak scaling, and will

try to make use of the entire system to solve a problem as large as possible. Hence the ability to distribute the computation across all available resources is of paramount importance in this domain.

Section 2.6 showed that distributing stencil computation is a complex task. The spatial dependencies between each point in the domain create overlapping inter-dependent sub-domains. The communication between the sub-domains has to be carefully balanced with redundant computation to optimize performance. In any case, a lot of traffic is generated, which can easily saturate the PCI buses interconnecting the GPUs for a large enough number of devices. The hardware specific implementation presented in this chapter exposes the non-uniformity of this communication channel and exploits it to improve performance. More specifically, the goals of this work are:

- to develop a high-productivity framework which preserves the portability of the OpenCL model and provides enough plasticity to implement known heterogeneous system optimizations. This shows that the hardware specific optimization techniques presented later in this chapter are orthogonal to existing optimization approaches and do not restrict the OpenCL model in any way.

- to understand, by experimental exploration, the influence of the inter-device communication structure upon the optimization space. In particular, how it affects the communication/redundant computation ratio. In order to address this rigorously, we generate synthetic, highly parameterizable kernels, informed by our analysis of a range of real examples. This makes our work robust in the face of future, and essentially orthogonal, improvements in the areas of single GPU optimizations and hardware performance, and also with respect to future variation in multi-GPU communication-computation ratios.

- to devise an autotuning heuristic, informed by our initial search space exploration, which is capable of selecting hardware specific settings for the various tuning parameters relevant to our chosen factors. Specifically, the heuristic will automatically determine the number of GPUs to use, the configuration of these GPUs with respect to the underlying system architecture, and the amount of overlap between the partitions of the domain.

The next section details the optimization strategies applicable to distributed stencil computation and their parameters.

5.2 OPTIMIZATION STRATEGIES

This section presents optimization strategies for distributed stencil computation and the different application characteristics affecting them. The key to efficient scaling lies in balancing the computation and communication overheads across the entire system. However, this ratio is affected by many factors, which have to be collectively optimized in order to achieve the best performance. To this end, the definition of stencil computation given in Section 2.6 is refined and some techniques inspired from prior works described in Section 3.3 are presented in more detail.

STENCIL PATTERNS Stencil computation is a well-recognized algorithmic skeleton in and of itself. However, the definition is broad and represents any nearest neighbor computation. We refine this definition here and present the different classes of stencil operators, which have distinct optimization strategies:

- *regular stencil*: this is the most common operator. It accesses neighbors in all dimensions of the domain.
- *irregular stencil*: a stencil operator is called irregular if it does not access any neighbor in at least one dimension.
- *composite stencil*: stencil computations are often defined as a sequence of individual operators. For example the sobel operator used for image detection is composed of two regular gradient operators combined by an irregular stencil.
- *iterative stencils*: many applications like physics simulations apply the same operator over and over again. Knowing how many iterations are applied in advance enables multiple timesteps to be computed in quick optimized succession.
- *converging stencils*: A variation of the iterative stencil does not have a known number of iterations, but provides a convergence function to stop the computation when a given criterion is met. These usually allow multiple timesteps to be computed between convergence tests.

Stencil computations can be defined as combinations of these patterns. The combined characteristics guide the choice of domain decomposition strategy.

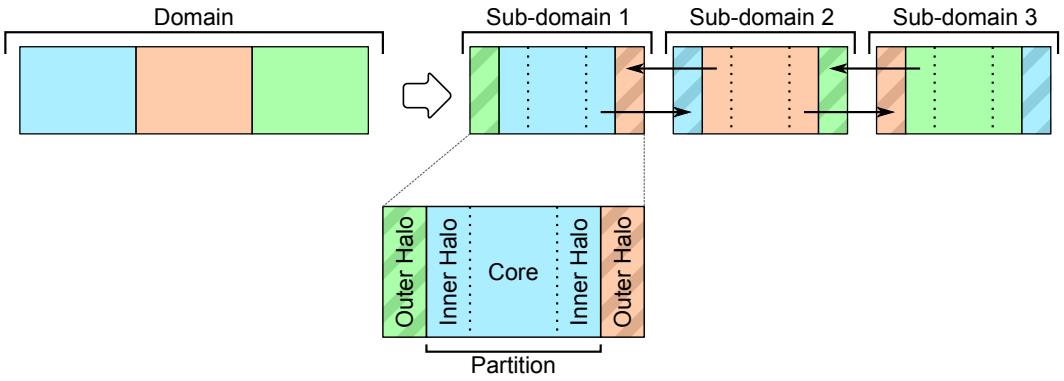
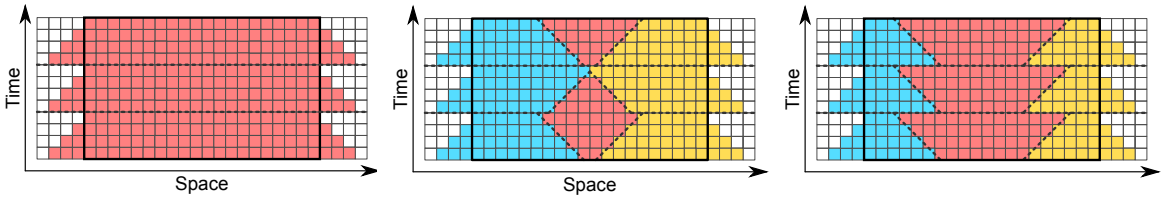


Figure 5.1: Example of decomposition of a 2D domain with warp-aound boundaries in 3 partitions. The tiles have to overlap to satisfy the dependencies at the edges of the partitions. This creates halo regions: the outer halo is the part duplicated from another tile and the inner halo is the part another tile depends on.

DOMAIN DECOMPOSITION Since computing each point in the space depends on neighboring values, cutting the domain in partitions creates broken dependencies at the boundaries. This can be resolved by creating overlapping tiles, as shown in Figure 5.1. The overlapping regions are called *halos*: the *outer halo* is a region from the edge of a neighboring partition which has been duplicated and the *inner halo* is the region from the tile which is copied by a neighboring partition, which is symmetrical to the outer regions along the cuts. The center part of the partition between the inner halos is called the *core* of the partition.

The minimal amount of overlap is the maximum stencil distance reached along the cut dimension in a single timestep; though increasing the amount of overlap enables the computation of multiple timesteps independently from the neighboring sections, thus decreasing synchronization and communication frequency. However, a greater overlap also introduces higher communication costs and redundant computation since the elements within the inner and outer halos are computed multiple times by independent partitions. Hence the size of the halo region has to be finely controlled: it has to be large enough to allow communication to be amortized by the core computation, but small enough so that it does not introduce excessive computation overheads.

DATA LAYOUT Arranging data in memory also plays an important role towards increased performance. Since communication between host and device is typically much slower than the memory bandwidth of the device, it can easily become the bottleneck. Exchanging only the halo regions between devices, as opposed to update the entire subdomains on the host, reduces the amount of data exchanged. However, it requires the halo



(a) Minimal Synchronization (b) Independent Halos (c) Maximum Overlap

Figure 5.2: Example of Domain decomposition strategies. Only cutting in the time dimension generates the least amount of synchronization (a). Space cuts need to be skewed to satisfy the dependencies in the middle of the tile (b). A higher priority can be given to the inner halos to maximize the chances of compute/swap overlap (c).

regions to be consecutive in memory. This can be achieved in two ways: either the cut is performed in the dimension used for linearization, since each slice is already consecutive, or the halos can be marshaled to a temporary communication buffer and unmarshaled after the swap. However, since we are targeting GPUs, scatter and gather operations to non-consecutive memory generates uncoalesced accesses which considerably increase the cost of these operations.

Data layout optimizations also have to take into account characteristics of the stencil operator. Many problems use distinct properties for each point in the domain, called fields. This introduces the array-of-structures (AoS) versus structure-of-arrays (SoA) dilemma introduced in Section 2.2.3.4, with the added complexity that different fields might get consumed at a different rate. SoA allows for fine grained control over each individual field to renew it only when it runs out. However, this generates a lot of communication and might introduce synchronization in a greater number of timesteps. AoS packs all the fields consecutively in memory and allows swapping to renew all of them at once, generating less frequent communication at the cost of a larger size. The drawback of this layout is that it also renews fields which were not entirely consumed, or fields which are not going to be read by the stencil operator.

LOCAL TILING & COMMUNICATION STRATEGIES The decomposition of the domain into tiles allows for computation of multiple timesteps until the halo region cannot satisfy the spatial dependencies anymore. The halos then need to be swapped with neighboring tiles. Figure 5.2 shows different strategies, which can be implemented by cutting the domain in the space and time dimensions. The three main strategies are:

- *Minimal synchronization*: Cutting only along the time dimension enables computation of the entire tile until the halo is consumed. The computation is then paused, and the outer halo is renewed. This triggers a synchronization every h/s iterations, where h is the halo size and s the maximum stencil distance along the dimension used for the cut. It also requires all devices to synchronize at the same timestep. This is the minimal number of synchronizations possible. However, the computation and communication never overlap.

- *minimal synchronization with independent halos*: splitting the tile in half enables computation and communication overlap: for example, the left-hand side of the tile can be computed first, then the left halo exchanged while computing the right tile. Alternating these schemes between each device spreads the communication over time since each half depends only on one neighbor. However, the center of the tile cannot be computed at the same time since it requires each side to be computed first. Four additional space cut spanning two swaps solves this dependency, so this scheme requires 6 synchronizations every $2h/s$ timesteps, which the minimal number of synchronization with independent halo possible.

- *Maximum Overlap*: a higher priority can be given to communication by computing the dependencies first. A space cut on either side of the tile, just wide enough to compute the inner halos, enables the swap to be initiated after a minimal amount of computation and amortize its cost with the core computation. This scheme gives the highest chance of copy/compute overlap since the minimum amount of computation is performed for the inner halos. However, it requires more frequent synchronization between each swap: the core of the tile depends on the inner halos and the time cut creates additional dependencies, generating 5 synchronizations every h/s timesteps.

Since these communication strategies have distinct properties, they could be different for different fields in the domain, creating many possible combinations.

The sheer amount of optimization combination makes implementing them by hand and exploring their parameter space highly impractical. The following section introduces Partans, a high performance and high-productivity distributed stencil computation framework abstracting these optimizations and automating their parameter tuning.

```
1  // Host/device structure definition
2  PARTANS_STRUCT( Pair, float f1; float f2; );
3
4  // Type aggregate definition
5  using TupFields = PARTANS_FIELDS( (Pair, a),(Pair, b),(float, err) );
6
7  // Volume allocation
8  Volume<TupFields> *volume = new Volume<TupFields>(800,600);
9
10 // Modifying a field in the volume at position (0,0)
11 volume->get<TupFields::a>(0,0).f1 = 0.5;
12
13 // Getting an element from the volume at position (5,0)
14 Pair t = volume->get<TupFields::b>(5,0);
```

Figure 5.3: Volume API in Partans. Line 2 declares a user-defined structure. Line 5 defines the type of each element of the volume, in this case two Pairs and a float. Line 8 creates a 2D volume of 800×600 elements. Line 11 and 14 demonstrate type safe element accesses.

5.3 THE PARTANS FRAMEWORK

The Partans framework is designed to abstract away the low-level implementation details that are required to perform stencil computation on multiple GPUs. It is implemented as a template library in C++. This language provides easy access to the OpenCL runtime – which is in C –, and enforces type-safety, in contract to the `void*` style used in the OpenCL runtime library. In addition, the template mechanism turns many customizations into compile-time decisions, as opposed to runtime overhead.

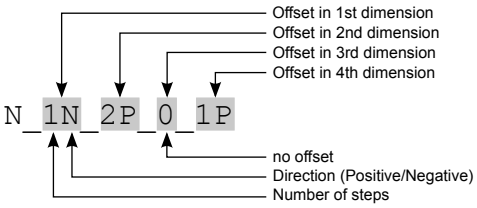
In this section, we describe the high level API, give an overview of the internal implementation strategy and present an overview of the tuning possibilities exposed by this design.

5.3.1 *API Concepts*

A stencil computation in Partans is characterized by three elements: a description of the volume representing the domain, a definition of an element function, which is the stencil operator, and a schedule definition controlling the mapping between the stencil operator and the domain.

```
1 void mult(global const Tuple *x,
2            global       Tuple *y,
3            global       float *err){
4   const float old = y[ORIGIN].f1;
5   y[ORIGIN].f1 =
6     (x[N_1P_0].f1+x[N_1N_0].f1+
7      x[N_0_1P].f1+x[N_0_1N].f1)/4.;
8   err[ORIGIN] = old - y[ORIGIN].f1;
9 }
```

(a) Stencil Operator



(b) Neighbor macro definition

```
1  Stencil<TufField> stencil{volume,
2     S::Converging|S::Composite};
3
4  stencil.addStencil(
5    cl_code, // element function
6    "mult",  // entry point
7    /*read: */{Tuple::a},
8    /*write:*/{Tuple::b,Tuple::err});
9
10 stencil.addStencil( cl_code, "mult",
11   /*read: */{Tuple::b},
12   /*write:*/{Tuple::a,Tuple::err});
13
14 stencil.addConvTest( 20,
15   R("bool test(global float *err){
16      return err[ORIGIN] < 0.01;})",
17   "test",
18   /*read:*/{Tuple::err});
19
20 // apply at most 1000 iterations
21 stencil(1000);
```

(c) Stencil object usage

Figure 5.4: Example of a stencil computation performing a Jacobi operator on one Tuple and storing the variation in the third field. The stencil operator (a) is defined as a plain text string and uses special macros (b) to access the neighbors. It is mapped to a volume using a stencil skeleton (c).

VOLUME DEFINITION The volume represents an aggregation of fields in a multidimensional domain. Each field can be a primitive type or a user defined type. Figure 5.3 shows an example of a volume definition. Line 2 is a declaration of a user-defined structure named Pair and containing two floats. Line 4 defines the type of the volume, which is a tuple containing a list of anonymous or named types. Line 8 shows an allocation of a two-dimensional volume of 800×600 elements. Each field can be accessed through a strongly typed getter by specifying the field name or index and the coordinates of the element in the volume.

STENCIL OPERATOR The stencil functor represents the computation at a single point in space. It is defined as a function operating over any number of fields, listing the readonly fields first and the writeonly fields second. Figure 5.4a shows an example of operator computing a blur function by averaging the four neighboring values of field x.f1 and storing the output in y.f1. The functor uses special macros to navigate in the space. These macros, described in Figure 5.4b, are composed of the prefix N_ followed by a list of positive or negative offsets in each dimension. For example, the macro x[N_1P_0] is equivalent to the 2D access x[1][0]. A special macro ORIGIN represents a null offset in all dimensions.

STENCIL SCHEDULE The mapping between the operator and the volume is controlled by a stencil skeleton. An example of stencil object declaration and usage is shown in Figure 5.4c. Users provide the main characteristics of the desired operation: in this case, the stencil will be applied to the volume *volume* and is a composite converging stencil, so multiple operators will be applied in succession and repeatedly until a condition is met.

Lines 4 and 10 add operators to the stencil computation. Similar to the OpenCL syntax, each operator is composed of a program, which is the operator function definition in a string, and the entry point of the function. The operator also takes a list of field masks representing the elements read from and written to by the operator. To avoid race conditions, an operator should never read and write at the same time, unless it is at the the origin, so the same field appearing on both sides triggers an error.

This syntax allows operators to be applied to a subset of the fields in the domain and decouples the stencil operators from the volume layout, allowing a high degree of operator reuse. In this case, both operators added in lines 4 and 10 apply the same functions, but with different parameters: the first call computes from a to b and the second from b to a, representing a double-buffered operation.

The convergence operator added in line 14 takes an integer representing the number of timesteps between tests, here the test will be applied every 20 iterations, and the rest of the parameters are similar to the stencil operators, except that a test function cannot modify any field.

Finally, the computation is invoked in line 21, with a limit of 1000 timesteps. Both operators will be applied iteratively and the convergence test will be evaluated every 20 iterations, until the tested values converge or the maximum number of iterations is reached.

This high level interface allows PARTANS to have full control over the memory layout, the generated device code, the decomposition and the communication; without being restrictive to the user in terms of expressiveness. Some underlying implementation details are briefly hinted in the next section and the tuning parameters they expose are described.

5.3.2 *Internal implementation strategy*

PARTANS performs a lot of low-level optimizations to improve performance, especially for generating the kernel functions and neighbor macros in an efficient manner. For example,

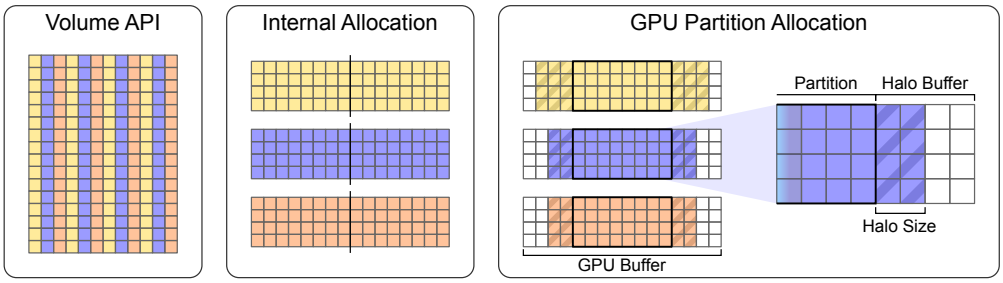


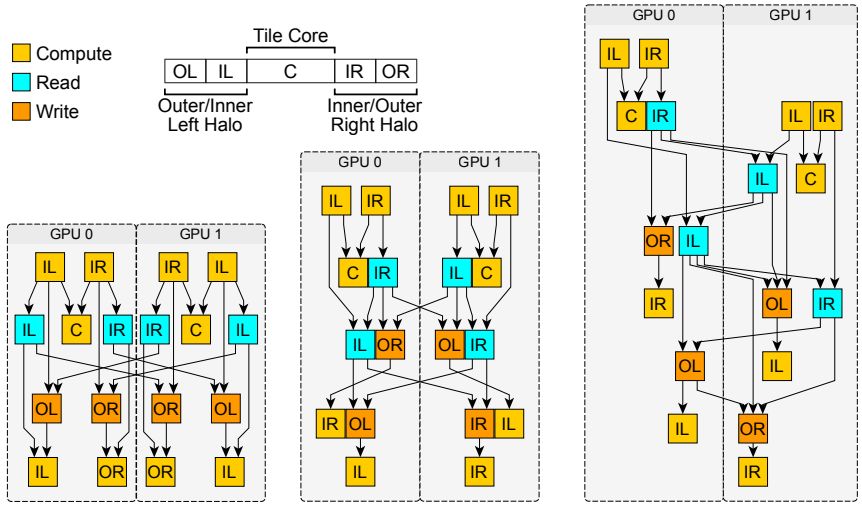
Figure 5.5: The PARTANS API provides an abstract interface for defining complex domains. Internally the fields can be interleaved or distinct. Each partition is copied to an oversized buffer, where the padding on either side enables halo to be resized dynamically.

modulo operations in index functions are replaced with bitmasks whenever possible to reduce arithmetic intensity and volumes are rotated according to the stencil shape to maximize memory coalescing. However, since the exploration focuses on multi-device communication, the single-device optimizations will not be described further. This section presents the implementation of the volume decomposition and the communication strategies affecting multi-device performance.

VOLUME ALLOCATION & TRANSFORMATIONS While programmers normally have to define a memory layout for data structures in C++, the type list used in PARTANS does not have this restriction. Decoupling types from layouts provides a strongly typed interface and lets the framework decide between SoA and AoS without affecting the codebase. Volumes are allocated lazily, using the information about the stencil operations applied to it.

A set of high level transformations such as padding and rotation can also be applied transparently since PARTANS controls all memory accesses in the stencil operator and the volume interaction. In Figure 5.5 for example, the original volume is switched to SoA and rotated 90 degrees to align the largest dimension. When the volume is rotated, all coordinates will be transparently transposed. This can be used to optimize the decomposition or make use of stencil operator properties such as irregular stencils.

DYNAMIC HALO RESIZING Memory buffers in OpenCL do not support dynamic resizing. However, we argued that finding the optimal halo depends on many factors and might change over the course of the application. To allow for more plasticity, the buffers on each device are allocated as large as possible, up to twice their original size, which creates a full duplication of the computation. As shown on Figure 5.1, the tile is copied at the center of this buffer, creating padding on either side called *halo buffers*. The framework can then



(a) Minimal Dependency (b) Device Duplex (c) System Duplex

Figure 5.6: Different communication strategies: by adding edges in the graph, Partans controls the communication parallelism. The memory operations can be independent (a), or limited to one read and write operation per device (b) or across the entire system (c).

dynamically change the size of the halo region within these buffers without affecting the tile itself.

TASK GRAPH To expose task parallelism, Partans generates task graphs to represent computation. For each volume partition, a set of inter-dependent kernels are enqueued for each tile to compute multiple timesteps, equivalent to each colored segment in each line for the strategies presented in Figure 5.2. Using dependencies between actions in OpenCL can express synchronization between devices implicitly: a swap is represented as a read operation on one device followed by a dependent write on another device.

Figure 5.6a shows an example of a simplified task graph for a single field using the *maximum overlap* strategy on two devices. When the outer halo runs out, the inner halos are read on either sides of the two tiles and written on the outer halo of the other device, after its inner computation finishes. These graphs get considerably more complicated for multi-field volumes with a SoA allocation or more devices, highlighting once again the importance of abstracting this from the user.

The swapping strategies can be fine-tuned by adding extra dependencies. While the task graph used in Helium aimed to generate the minimal number of dependencies, Partans introduces redundant dependencies to enforce an evaluation order. The insight of Partans

is that since underlying hardware resources are shared (here PCIe buses), exposing too much data parallelism might result in a congestion and be sub-optimal.

Figure 5.6a presents a task graph with the minimal set of dependencies for a single field. Extra dependencies are added to the graph in to enforce duplex communication at a device level (Figure 5.6b), or in the entire system (Figure 5.6c). These different strategies affect the strain put on the communication channels and their efficiency depends on system resources.

5.3.3 *Optimization space*

The optimization process within PARTANS sets out to exploit hardware specific behavior to enhance throughput in a multi-device system. To preserve portability, this process is completely oblivious of explicit hardware specifications and user intervention. The method implemented in the optimizer uses direct observation of the hardware interference on the tunable parameter space. Hence, it is essential that PARTANS controls and explores as many tuning parameters as possible to identify hardware bottlenecks.

The vast majority of the factors affecting performance are abstracted by the high level API. Its design successfully decouples the specifications of a stencil computation from the internal implementation. This in turns creates a clear separation of concern between the user, who can focus on expressing the problem at a high level, and the framework, which implements application independent optimization strategies. More specifically, the main parameters under PARTANS' control are:

- *halo size*: the best halo size may vary at runtime. Using the halo buffers, a dynamic search can be performed for each phase, without re-allocating and initializing buffers.
- *data layout*: the memory representation of a domain can take many forms. Fields can be independent or interleaved. Transformations such as padding or rotation can be applied transparently to data placement: the framework is free to decide how many times the domain should be divided and how tiles are mapped to devices.
- *communication strategies*: tweaking communication dependencies between the tiles enforces specific patterns which control the amount of parallel communication. This technique can be used to implement a strategic ordering or limit the contention.

All of these parameters are sensitive to communication/computation ratios, hence they are all interdependent. Establishing the interference between these parameters through exploration is necessary before synthesizing them in an automatic tuner. The following section introduces the setup used for the experimentations.

5.4 EXPERIMENTAL SETUP

To explore the impact of hardware artifacts on the scalability of stencil computation, a test suite comprising a variety of application domains was implemented. These benchmarks are evaluated on different multi-device systems, in which all devices are programmatically indistinguishable but subtle architectural differences exist. The benchmarks and systems are presented in this section.

5.4.1 *Benchmarks*

Real world stencils involve highly diverse dimensionalities, shapes and volumes. As the investigation focus is on communication optimization, the chosen set of six applications all apply stencils iteratively, meaning that halo swapping is necessary to complete the computation. Our benchmark suite consists of the following programs:

- *Game of life* is a cellular automaton implementing Conway's Game of Life. Every iteration represents a new generation of the simulation. The application uses double buffering, meaning that each element of the volume is composed of two chars.
- *ReverseEdge* is an image-processing application using a Jacobi operator. A one-way function, edge detect, is applied on an image, and a Jacobi operator is used to approximate the original image.
- *Swim* is a fluid dynamics kernel used for weather prediction. It is adapted from the SPEC OMP 2001 benchmark suite. The fields are composed of complex structures and there are three different operators involved in the computation.
- *Himeno* is a fluid dynamics application computing a Jacobi-variant converging stencil. It uses complex fields to represent the main operator of the Poisson equation used in mechanical engineering and theoretical physics.
- *Hyperthermia* is a simulation of the temperature diffusion in human bodies during hyperthermia cancer treatment.
- *Tricubic* is a 64-point stencil used in numerical analysis for tri-cubic interpolation.

Benchmark	Dimensions	Stencil points	Fields	Reads	Writes	Flops/Point
Game of Life	2D	9	2	9	1	n/a
Reverse Edge	2D	5	3	5	1	5
Hyperthermia	3D	6	11	17	1	16
Himeno	3D	27	15	25	2	33
Swim	2D	9	13	28	11	63
Tricubic	3D	64	5	64	1	132

Note: Game Of Life uses only bit masking on integer type.

Table 5.1: Summary of benchmark characteristics. The dimensionality define the volume. The other properties apply to the stencil operator. Each property is given in number of fields rather than their size in bytes.

Table 5.1 presents the stencil characteristics of each application. The operators are defined by the dimensionality of the volume, the stencil shape, the number of fields (total, read from and updated) and the number of floating point operation in the element function. The boundary condition has been set to wrap-around for all the benchmarks, which forces halo consumption on both sides of the partition and creates a homogeneous amount of computation across all tiles. A simpler boundary policy, like Dirichlet boundaries, effectively halves the communication in the case of two partitions, as there is only a single dependency instead of two.

The net impact of communication optimizations on a whole application evaluation is affected by the performance of the computational phases between synchronizations. Faster execution of computation magnifies the relative importance of fast communication. In the context of GPU implementation of stencils, quicker computation phases could arise from local computational optimizations, or simply from larger or more powerful GPUs. For example, speedups of orders of magnitude have been reported in the literature using single GPU optimization strategies, and compute power of GPUs currently increases at a faster rate than their communication capabilities [1].

In order to extend the value of the analysis against such developments, the test suite is augmented with virtual variants of each benchmark. The consumption rate of each halo is preserved, but the actual computation is replaced by synthetic workload for which the granularity is accurately controlled. This enables the exploration of a fuller space, with kernel execution time ranging from that of the original application, to faster versions that may emerge as a result of the trends above.

1 In the same time that PCIe bandwidth has doubled (PCIe 2.0 to PCIe 3.0), the average compute power of GPUs has tripled.

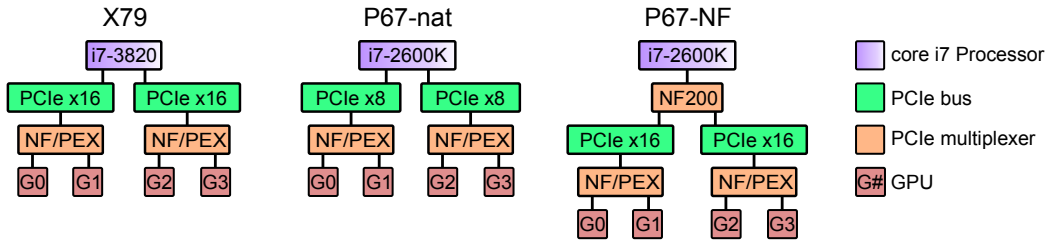


Figure 5.7: Overview the four evaluation systems. The P67 system offers two different layouts for dual GPU configuration: either two PCIe x8 in native mode or two PCIe x16 via a NF200 multiplexer. Both the Nvidia GTX 590 and AMD Radeon 5970 are dual GPUs graphics cards that include a PCIe multiplexer on their PCBs.

5.4.2 *Architectures*

This section describes the hardware used to carry out the experiments. Since the communication is being investigates, the PCIe layout is more relevant than the compute power characteristics of the GPUs. PCIe was introduced briefly in Section 2.2.2.3, which explained that the number of GPUs in a system can be increased by introducing multiple PCI root complexes and multiplexers. This creates a network where the end points have identical characteristics, for example 16 lanes, but the underlying topology might be irregular. This layout is defined by the motherboard chipset and the graphics cards' design and is completely transparent to the user. However, our experimentation shows that it affects performance.

MOTHERBOARDS & SYSTEMS Three different systems are used. Our first system uses an Intel X79 motherboard chipset with an Intel core i7-3820 CPU, which supports two native PCIe x16 slots. The system runs Linux with a 3.1.10 kernel. This setup is referred to as *X79*. The second system uses an Intel P67 motherboard chipset with an Intel core i7-2600K CPU, which supports two native PCIe x8 slots. The system runs Linux with a 2.6.37.6 kernel. This setup is referred to as *P67-nat*. The third system is identical to the second one, except that the two PCIe x16 slots that are connected to the core i7 processor via an Nvidia NF200 PCIe multiplexer, offering two PCIe x16 slots. This setup is referred to as *P67-NF*. PARTANS always uses pinned memory in the host in order to optimize transfer time over PCI.

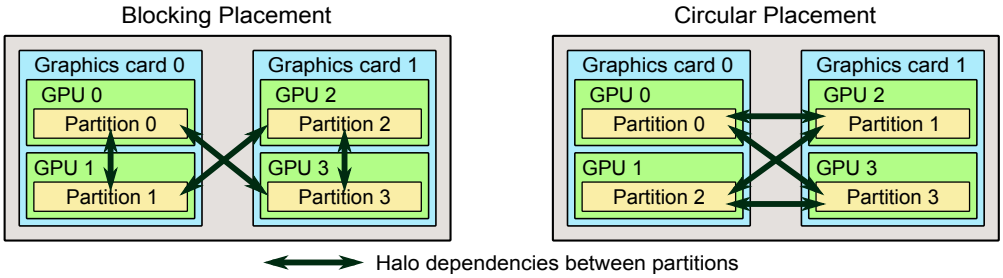


Figure 5.8: Different partition dispatching strategies for a four GPU setup. The arrows represent semantic dependencies between neighboring partitions.

GRAPHICS CARDS Our setup uses two dual GPU cards. The AMD Radeon 5970 has two TeraScale 2 GPUs sharing a single PCIe slot through a PEX multiplexer. The Nvidia GTX 590 has two Fermi based GPUs and uses an Nvidia NF200 multiplexer. The OpenCL runtimes are provided by CUDA SDK 4.1.28 and AMD APP SDK 2.6.

The three different configuration types with the attached graphics cards are shown in Figure 5.7. The bottom multiplexers are found on the PCB of the Nvidia and AMD graphics cards, and the multiplexer splitting the PCIe lanes is located on the motherboard. No processor or graphics card are overclocked. In total, the three systems and two GPU types provide six different setups to evaluate. Since the evaluation focuses on seemingly homogeneous systems, the configurations using graphics cards from different vendors in the same system were not investigated

DUAL GPU PLACEMENT When distributing data to two GPUs, the setups offer two possible configurations. The data can be assigned to GPUs on the same PCB (e.g. Go and G1 in Figure 5.7). This configuration is called *Single Card*. Alternatively, the data can be assigned to GPUs on different PCBs (e.g. Go and G2). This configuration is called *Dual Cards*. Unlike CUDA, OpenCL currently does not support direct device-to-device copy, thus all the communication is centralized and has to go through the host memory.

FOUR GPU PLACEMENT When using all the available GPUs, there is still a choice to be made on data placement. Figure 5.8 illustrates the two strategies: *blocking placement* or *circular placement*. Blocking placement first assigns partitions to one dual GPU, and then moves on to the next one. This configuration is called *Blocking*. Circular placement assigns partitions in an alternating way. This configuration is called *Circular*. The figure also shows the difference between the dependencies. The arrows in this case indicate the dependencies between the partitions, not the actual communication.

5.4.3 *Evaluation Methodology*

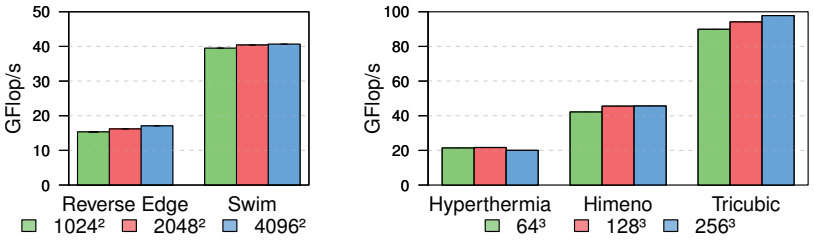
In all experiments, the runtime is measured using wall-clock timers in the host program. Only the main stencil loop is timed for 10,000 iterations. This values is used to compute the average time for one iteration. Because the GPU measurements can be noisy, due to frequency scaling or shared resource utilization, multiple sampling techniques are used to ensure the stability of the results. For each sample, the main iterative stencil loop is applied 16 times in a row with the same number of iterations. If the margin of error for the 95% confidence interval in the interquartile range is below 5%, the sample is considered successful and the truncated mean of the set is stored. When exploring a parameter range, the space is sampled in random order until five samples are accumulated for each point. This guarantees sampling of similar parameters will be scattered in time to reduce temporal variation of the GPU performance due to external parameters. The final result is the median value of the five interquartile means.

5.5 EXPERIMENTAL EVALUATION

5.5.1 *Overview*

This brief summary highlights the main findings, which are discussed in greater detail in the subsequent sections. Section 5.5.2 investigates the absolute performance obtained by the single GPU implementation, which acts as a baseline for the subsequent speed-up results. The single device performance shown to be competitive with previously published work on similar applications and devices. For example, PARTANS obtains around 70% of the performance of the highly tuned PATUS framework [CSB11].

Turning to the multi-GPU implementation evaluation, Section 5.5.3 shows that the relationship between halo size and performance has a regular bitonic form across all cases. This creates an identifiable sweet spot, whose precise location varies with application and system. Similarly, problem size has a relatively simple relationship to performance (larger problems have a greater potential), while halo shapes which go beyond trivial nearest neighbor can produce less intuitive effects. Finding the correct settings can result in improvements in speed-up of the order of 50%, compared to other points which are quite close in the optimization space.



(a) 2D Stencils (b) 3D Stencils

Figure 5.9: Absolute single GPU performance on an Nvidia GTX 590 in GFlop/s for different volume sizes. The graph also shows 99% confidence intervals, however these are so small that they are almost invisible.

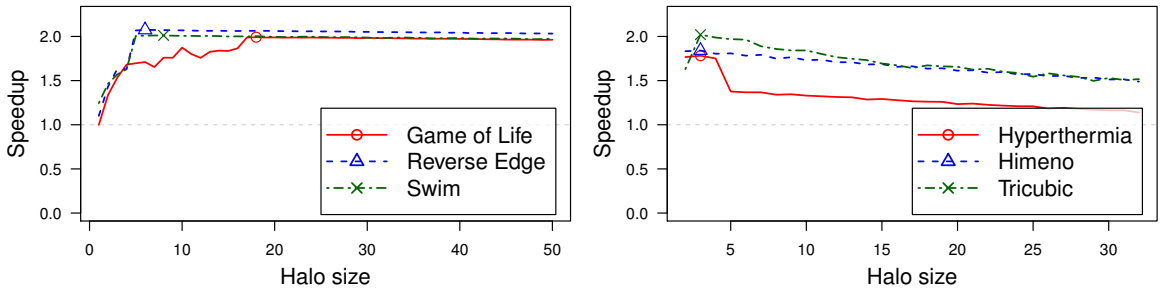
Section 5.5.4 explores the impact of the underlying communication technology on otherwise homogeneous systems. For situations in which a subset of devices is selected, speedup variations up of up to 33% can be observed in regions of the search space at or close to the optimal settings for halo size. A correct selection of devices also varies with problem granularity – with a performance improvement of up to 13% at stake. For experiments using all four available GPUs, the optimal allocation of partitions to devices is dependent upon halo size and compute granularity – with discrepancies of up to 21% and 13% in favor of *Blocking* and *Circular* respectively, at different points in the space. These findings all serve to emphasize the challenge involved in the full autotuning problem.

The autotuning strategies are evaluated in Section 5.5.5. These behave impressively, typically obtaining over 90% of the performance improvements found by an exhaustive search of the space.

5.5.2 *Single GPU performance*

Figure 5.9 presents the raw performance for each application using a single GPU on the GTX 590 card. These results are comparable to those obtained by Phillips and Fatica [PF10], who implemented an optimized version of the Himeno benchmark on Nvidia Tesla C1060, and Matthias Christen [MC11] who implemented Hyperthermia and Tricubic benchmarks using their own PATUS framework on an Nvidia Tesla C2050. In direct comparison to the Tesla C2050[2], our naïve implementation achieves about 70% of the performance of the highly optimized PATUS. We did not try to close this gap any further, as the focus of Partans' evaluation is multiple GPU optimizations – which PATUS does not offer.

2 The Tesla uses the same GF110 graphics chip as the GTX 590, but clocked about 5% slower.



(a) 2D Stencils - problem size: 4096 × 4096

(b) 3D Stencils - problem size: 256 × 256 × 256

Figure 5.10: Impact of halo size on performance. The problem size is in all programs is 2^{24} grid points and is distributed across 2 GPUs. On each graph, we mark the optimal halo size that obtains the best performance.

The figure also shows that raw performance for each application is not constant across volume sizes. In order to point out trends and application independent observations, the remainder of the experimentation reports speedup over single GPU performance instead of raw performance.

5.5.3 *Halo Size Impact*

Section 5.3.2 described how varying the halo size affects the balance between data communication and redundant computation. We now explore which application parameters have an impact on the optimal halo size. The platform used in this subsection is the P67-NF setup with one GTX 590 dual graphics card. The speedup measured is compared to the performance obtained by using only a single GPU of the GTX 590 cards.

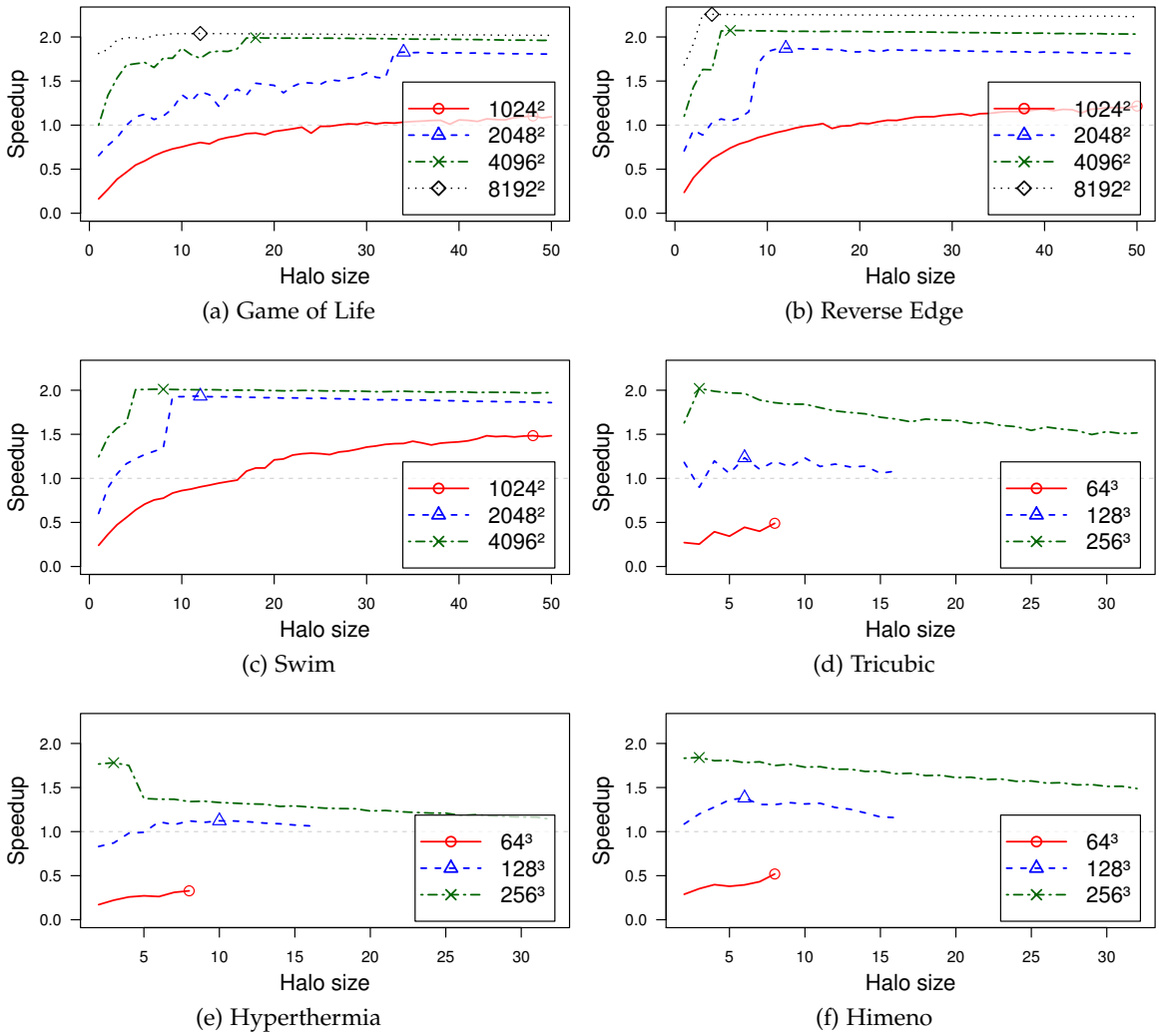
VOLUME DIMENSIONALITY Figure 5.10 shows the impact of the halo size for several applications in two and three-dimensional space. Both spaces have the same number of grid points (2^{24}) spanning a square or cube, respectively. We observe for all curves two distinct stages: an ascending and a descending phase. When the halo size is small, the swapping frequency is increased and the latency induced by initiating the swap plus the communication cost itself cannot be hidden by the computation of the core. As the halo size increases, the swapping frequency decreases, leading to less overhead and thus an increase in performance. With increasing halo size, the amount of computation also increases to compute the elements in the outer halo, leading to a steady slowdown. This effect is more noticeable for 3D applications (see Figure 5.10b), because each increment in halo size adds

an additional plane that needs to be computed, in contrast to a vector for 2D problems. The sweet point between the two phases is the optimal halo size. This spot is marked on each graph. In general, we notice that the dimensionality of the application affects the basic shape of the halo performance curve and location of the best halo. The benchmark *Game of Life* has an optimal halo considerably larger than the other 2D applications. This benchmark uses a single 8 bit field, requiring much less space and allowing faster exchange.

PROBLEM SIZE The second investigation evaluates the impact of the problem size on the optimal halo size. Figure 5.11 shows that the optimization space becomes more complex as the input size varies. The speedup is around 100% for large problem sizes in all applications. However, for smaller problem sizes, the scalability is not as good. The smallest problem size (1024×1024) for 2D applications only has a speedup of 27% on average. The curve just shows the increasing phase, indicating that the bottleneck is still communication despite very large halo regions. 3D applications show the weakest scalability for small problem sizes, as communication is even more expensive. Furthermore, for a cubic input of size 64, the maximum halo possible is too small for the communication to be amortized.

STENCIL SHAPE *Tricubic* shows an interesting impact of the stencil shape. All other applications have stencil shapes that access only the nearest neighbors. However, *Tricubic* also accesses neighboring points with a distance of two. Thus increasing the halo size from an even number to an odd number results in an increase of redundant computation, but still requires the data exchange to happen at the same frequency as before. This creates the sawtooth pattern visible in Figure 5.11d, particularly for problem sizes of 64^3 and 128^3.

DEVICE CAPABILITIES Figure 5.12 shows the impact of the hardware performance on both the halo size and the scalability for a selected range of applications. The overall capabilities of the Radeon card are lower than the GTX in terms of compute power and bandwidth. The scalability of each problem is completely different across the two devices: all three applications scale linearly on the GTX, but only *Game of Life* exceeds a 75% speedup on the Radeon. *Tricubic* even suffers a considerable slowdown on the Radeon. For all applications, the optimal halo sizes are considerably larger on the Radeon than the GTX.

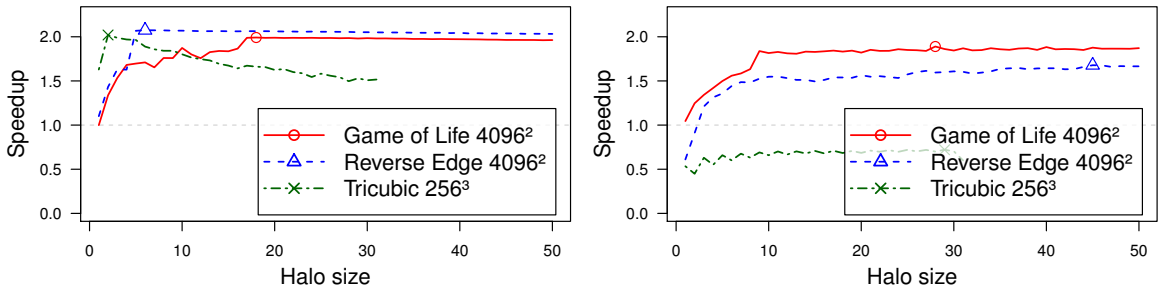


(a) Game of Life (b) Reverse Edge

(c) Swim (d) Tricubic

(e) Hyperthermia (f) Himeno

Figure 5.11: Impact of the problem size on the optimal halo size. The legend shows the number of grid points in the total volume. On each graph, we mark the optimal halo size that obtains the best performance. For the 3D application, the small volumes are too small to investigate larger halos.



(a) Nvidia GTX 590 (b) AMD Radeon 5970

Figure 5.12: Comparison of hardware performance between Nvidia GTX 590 and AMD Radeon 5970. The devices do not have enough memory to support a larger halo size than 32 for *Tricubic*.

5.5.4 *Data Placement and PCIe Layout*

This section presents an experimental exploration of the influence of the partition mapping to multiple GPUs in seemingly homogeneous systems. This experiment demonstrates that the underlying PCIe layout has an influence on the performance in various aspects, including the optimal halo size and scalability. Therefore, it is an important tuning parameter.

We first explore the impact of the device choice for applications using a subset of available GPUs, and then measure the impact of data locality when using all available resources.

5.5.4.1 *Dual GPU exploration*

Utilizing all the GPUs available is not necessarily the optimal solution. For example, the problem size may be too small to be decomposed into many parts, or the communication costs involved may make applications unable to scale efficiently to more than two GPUs.

This section presents the performance impact of choosing different device subsets in setups providing four GPUs split over two cards, as described in Section 5.4.2. In order to explore the space fully, and because the ratio between compute and communication performance of future GPUs is hard to predict, it is explored by artificially shrinking the time of the compute operation from the time taken by our naïve implementation to virtually nothing. The *compute granularity* is defined as a fraction of the naïve runtime, where 100% is the time taken by our unoptimized version and 0% represents an instantaneous computation. This allows us to investigate the space generated by applications which have had their single GPU kernels more highly optimized or which are running on more powerful hardware. As in the previous section, the impact of the halo size is explored for each compute granularity to see the impact of the compute time on the communication overhead.

DIFFERENCE MAPS Figure 5.13 shows the direct comparison of two different configurations running on the same system, the *P67-NF* with two dual GPU GTX 590 graphics cards, but using different PCIe layouts. The left graph shows the *Single Card* configuration, while the right graph shows the *Dual Cards* configuration. In both cases, two PCIe multiplexers (one on the motherboard and one on the graphics card) need to be traversed in order to communicate with a GPU. Both plots show the *Himeno* synthetic kernel with an input size of 256^3 grid points. The lines plotted on each of these graphs represent the optimal halo size, which is the halo size giving the highest speedup for a given granularity.

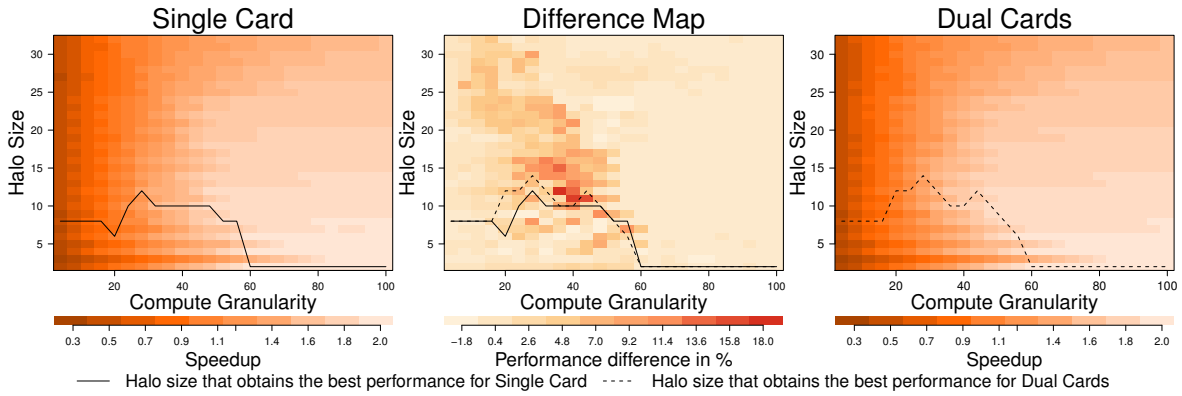
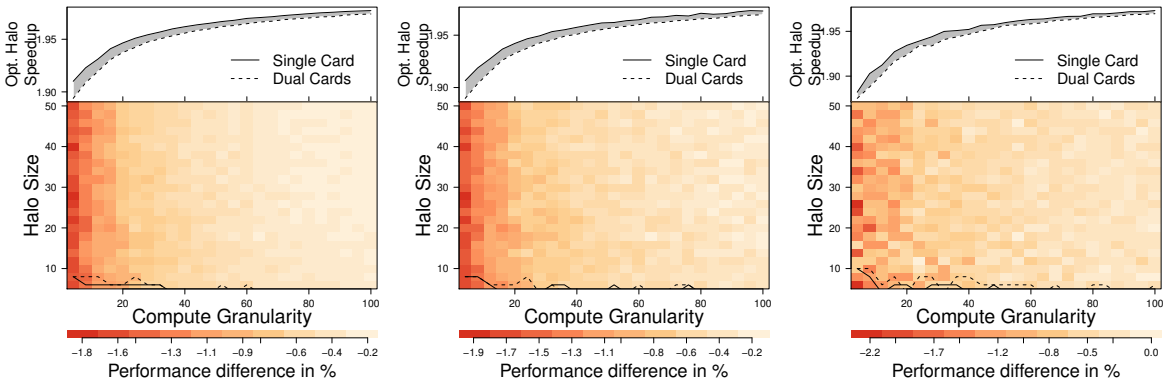


Figure 5.13: Comparison of PCIe layouts. A synthetic kernel is run using two GPUs in a four GPU system (using two GTX 590). The left graph shows the speedup achieved using the *Single Card* configuration when varying the halo size and compute granularity. The right graph shows the speedup using the *Dual Cards* configuration. The middle heatmap represents the speedup difference between the second and the first configurations.

The middle graph shows the performance difference of speedup between the two configurations. This plot highlights the performance difference in the full space; note that while both speedup graphs look very similar, the difference map reveals a performance difference of up to 18%. The optimal halo lines on the difference map are identical to the ones on the left and right plot. They illustrate whether the optimal halo size traverses regions with high differences in performance. In this case, the two configurations show a speedup difference of up to 16% for an optimized halo size in favor of the *Dual Cards* strategy.

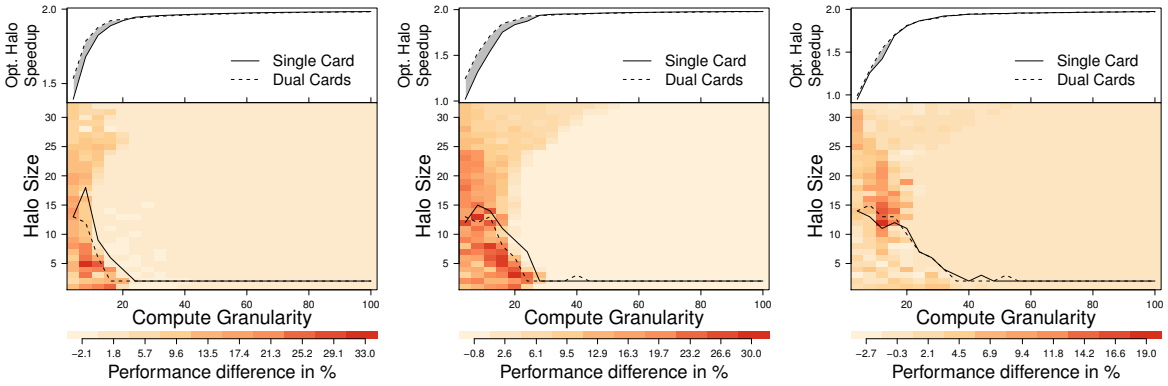
In the rest of this section, we only use difference maps to visualize the difference between the *Single Card* and *Dual Cards* configurations. We summarize the speedup information as a line plot above the difference map (see Figure 5.14a for example), which shows the best obtainable speedup for a given compute granularity. All evaluations are performed using the Nvidia GTX 590 graphics cards unless stated otherwise – since they have better hardware performance than the AMD Radeon 5970 and hence exarcebate the communication pressure. Figures 5.14 to 5.16 show the difference maps for different synthetic kernels running on our three evaluation systems.

OPERATOR IMPACT For the 2D synthetic kernel *Reverse Edge* (see Figure 5.14), there is very little difference between the *Single Card* and *Dual Cards* configuration. For small compute granularities, the *Single Card* configuration tends to consistently obtain about 1.5% higher speedup than the *Dual Cards* configuration. The similarity between the three systems is caused by the high scalability of the two-dimensional problem. The speedup achieved



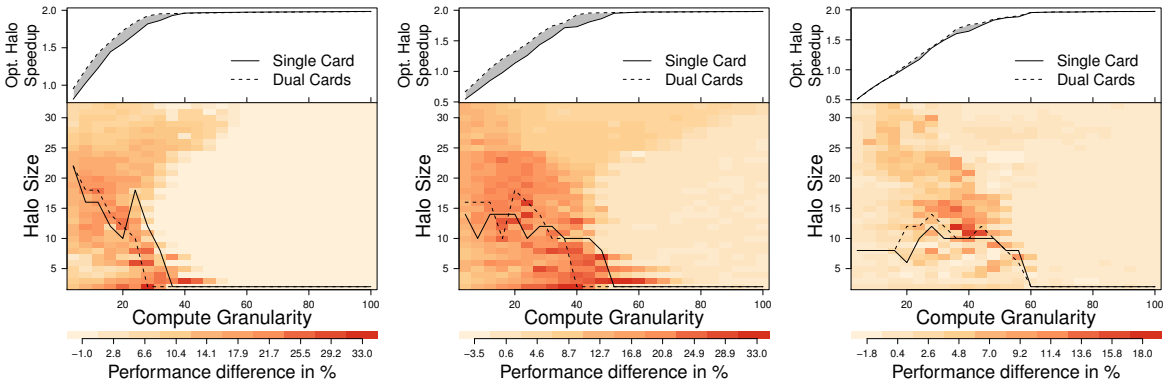
(a) *X79* (b) *P67-nat* (c) *P67-NF*

Figure 5.14: Impact of the PCIe layout on different systems using the GTX 590 GPUs for *Reverse Edge* with an input size of 4096^2 grid points.



(a) *X79* (b) *P67-nat* (c) *P67-NF*

Figure 5.15: Impact of the PCIe layout on different systems using the GTX 590 GPUs for *Hyperthermia* with an input size of 256^3 grid points.



(a) *X79* (b) *P67-nat* (c) *P67-NF*

Figure 5.16: Impact of the PCIe layout on different systems using the GTX 590 GPUs for *Tricubic* with an input size of 256^3 grid points.

by using two GPUs over one GPU tends towards 100%, and the lower limit for the entire space is above 90%. We find similar results for the other 2D synthetic kernels.

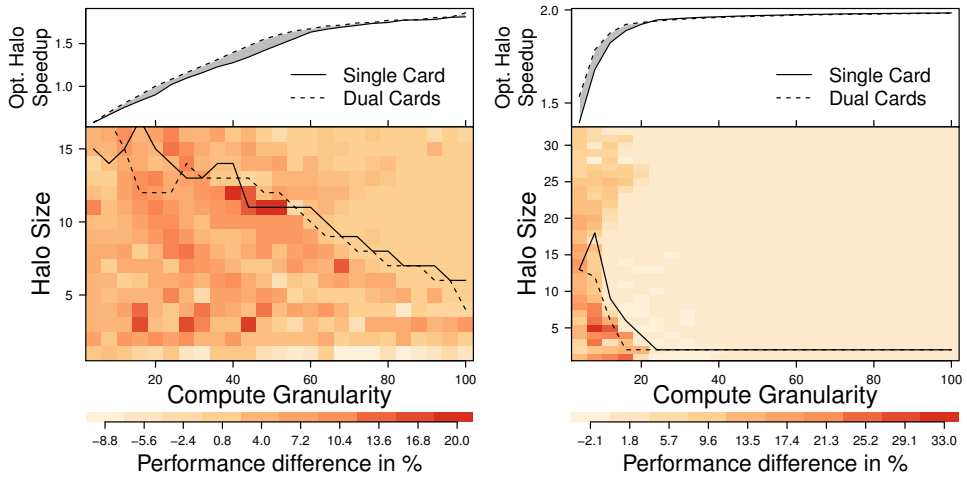
Figure 5.15 shows the same experiment for the *Hyperthermia* synthetic kernel. This space is very different. For the *X79* and *P67-nat* systems (see Figures 5.15a and 5.15b), which are both using native PCIe connections, the overhead of the multiplexer is clearly visible for small compute granularities using the *Single Card* configuration. The rest of the space shows little difference between the two configurations: this corresponds to the area where the communication is completely amortized by computation. The communication overhead starts taking over gradually as the granularity decreases. This happens either when the halos are too small and have to be swapped too often to be amortized by the computation, or, when they are too large and the swap takes too long to be hidden by the core computation. This explains the round shape of the lightly colored area along the left side.

For the two first systems, we can clearly observe the gain of avoiding the multiplexer. For the *Single Card* configuration, communication has to go through the on-GPU multiplexer. In the *Dual Cards* configuration, both GPUs are accessed through the on-GPU multiplexer. However, as the second GPU on each card is idle, the overhead is negligible or null. The *Dual Cards* configuration obtains higher speedups for lower compute granularity. This difference is accentuated in the *P67-nat* system that only provides PCIe x8 links, compared to the PCIe x16 links found in the *X79* system.

The multiplexer becomes unavoidable in the *P67-NF* system for both configurations. In total, there are three multiplexers: one on the motherboard and one on each graphics card. For the *Single Card* configuration, one graphics card is not being used but the multiplexer on the other one is used to access both GPUs. For the *Dual Cards* configuration, one GPU on each PCB is idle, but the multiplexer on the motherboard is being used to access both graphics cards. Our experimental exploration shows that in this case using both cards is still beneficial, with a gain of up to 22.69% along the optimal paths.

Figure 5.16 shows the same experiment for the *Tricubic* synthetic kernel. We observe a similar behavior as for *Hyperthermia*. However, the larger stencil shape in *Tricubic* results in more communication and increases the difference between the two configurations further.

PROBLEM SIZE Figure 5.17 shows the impact of the problem size on the performance difference for *Hyperthermia* running on the *X79* system. As established in Section 5.5.3, modifying the problem size changes the communication pressure. This extra pressure on



(a) Hyperthermia 128 (b) Hyperthemia 256

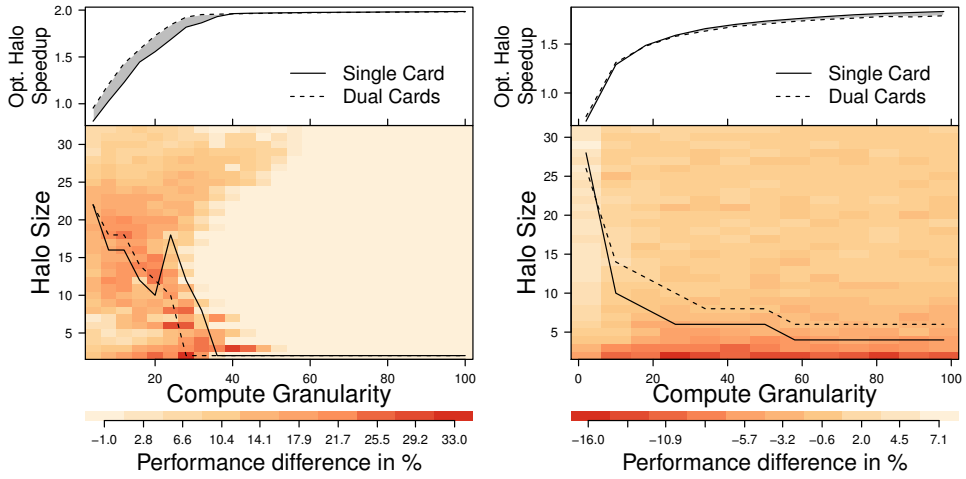
Figure 5.17: The impact of the PCIe layout depends on the problem size. (a) presents the *Hyperthermia* synthetic kernel on the X79 chipset with a problem size of 128^3. (b) shows the performance difference for the same conditions but a problem size of 256^3.

the smaller volume translates into a performance difference across the entire space, in contrast to no difference for most of the space when using a larger volume.

DEVICE Figure 5.18 shows a selected result comparing the performance difference using different GPUs. Figure 5.18a shows the difference map for *Tricubic* running on the *X79* system using the GTX 590 graphics cards. Figure 5.18b shows the difference map for the same setup, except that we replaced the graphics cards with AMD Radeon 5970. Besides the scalability difference, which is caused by the inferior compute performance of the Radeon 5970, we notice that the optimal configuration depends on the compute granularity. For most of the presented results, the *Single Card* configuration is the best choice for the Nvidia GTX 590 GPUs. For the Radeon 5970, the *Dual Cards* configuration performs best for small compute granularities, while the *Single Card* configuration performs consistently better for larger compute granularities. Choosing to use a single card leads to a performance improvement of up to 13.2% along the optimal path.

5.5.4.2 *Data Placement for Full System Utilization*

Figure 5.19 is similar to Figure 5.13, but this time all the available GPUs are being used. Both configurations show a similar speedup, up to 3.85x for high compute granularities. However, the maximum observed slowdown also increases compared to using just two



(a) Nvidia GTX 590 (b) AMD Radeon 5970

Figure 5.18: Impact of the PCIe layout on the *X79* system using different graphics cards for the *Tricubic* synthetic kernel with an input size of 256^3 grid points.

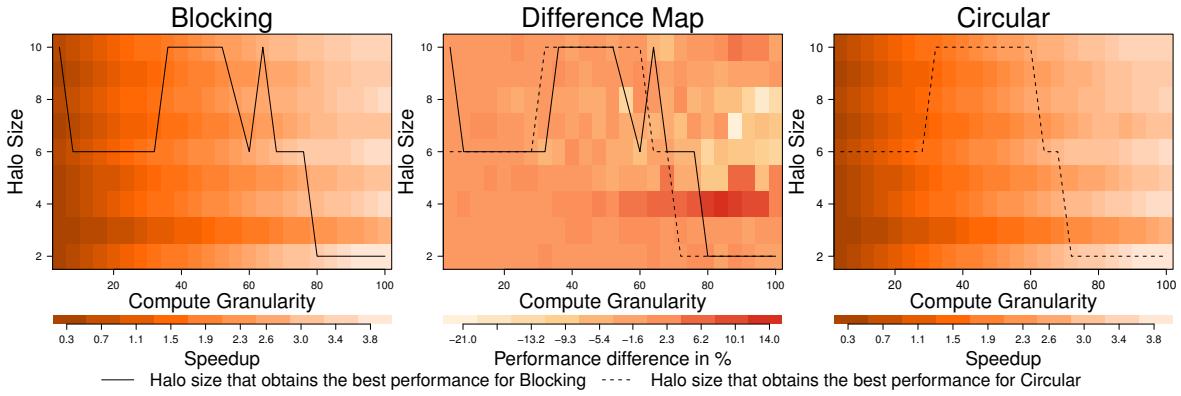


Figure 5.19: Data placement impact when using four GPUs. The left plot shows speedup when four partitions have one dependency on the same card and the right hand side shows another placement where partitions have both dependencies on the other card.

GPUs. Having four partitions increases the pressure on the swapping cost, and all the communication involved is harder to amortize by the core computation.

Even though the full system is utilized, the difference map shows some inequalities between the configurations for lower compute granularity. Over the complete space, *Blocking* is up to 21% faster than *Circular*. At other points in the space, *Circular* is up to 13% faster than *Blocking*. The optimal path for both configurations crosses this space, meaning the best partition mapping decision depends on the compute cost in this case.

5.5.5 *Autotuning*

Our results so far confirm the non-trivial nature of the stencil optimization space, and hence the need for an automated approach to its navigation. In this section, we describe and evaluate our autotuning strategies. Our strategies combine offline and online phases.

The first offline decision concerns the *volume orientation*, as explained in Section 5.2. This optimization simply requires examination of the volume and stencil shape. In the case of an asymmetric stencil, the volume is aligned to minimize the swapping frequency. Otherwise, the volume is aligned to be cut along the largest dimension, which allows a broader range of halo sizes to be considered and minimizes the cut area.

In the next offline step, we select a swapping strategy from a pool of predefined strategies (presented in Section 5.2). Our experiments so far indicate that prioritizing computation of the inner halos is optimal in all cases; hence we select and implement it offline. However, our framework can easily switch this into an online decision should the need arise.

The framework then tunes the *GPU selection and partitioning*, which is an offline decision. It determines how many, and which GPUs to use, and in the case of full system utilization, how to assign partitions to GPUs. To achieve this, all stencil operators used in the application are automatically extracted and profiled independently offline. Using a fine grained granularity allows PARTANS to measure the communication interference between the layouts. A medium granularity is used to assess the scalability of the stencil independently of the optimum halo size. The results are combined, weighted by the complexity and usage of each stencil in the case of multi-operator applications, to make an overall decision.

Finally, the *halo size* is adapted online. As the application runs, we vary the halo size and gather performance data. The data is used to refine the halo adjustments. We have experimented with a range of adjustment strategies. The search is always informed by the

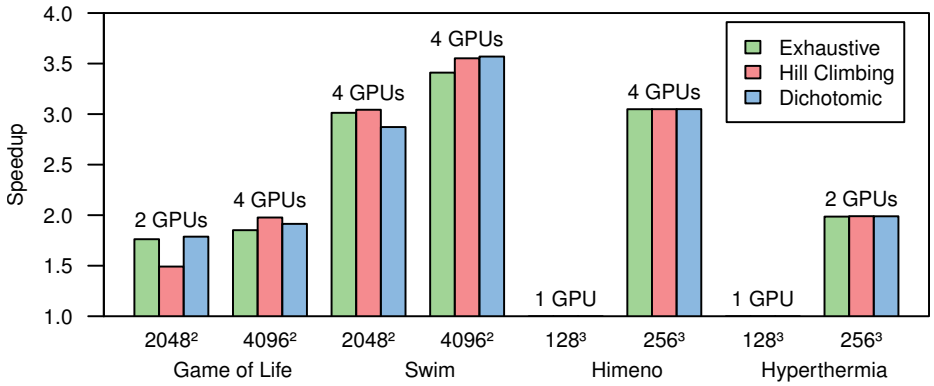


Figure 5.20: Speedup over single GPU using autotuning for two input sizes and 100,000 iterations, including online tuning overheads. The autotuner decides how many and which devices to use and performs an online search for the optimal halo size using several search strategies.

(application specific) shape of the stencils to prune part of the search space. This avoids the oscillating performance for applications like *Tricubic* (discussed in Section 5.5.3).

The simplest is an *exhaustive search*: try all feasible halo sizes in order and eventually pick the best. The second strategy is inspired by the observation that performance has a simple bitonic relationship to halo size. The *hill climbing search* increases the halo size linearly, from the minimum, until performance degrades for several consecutive points.

These two strategies are linear searches, resulting in poor performance if the optimal halo is in the middle of the range or at the opposite end. To guarantee reasonable convergence time, we also implemented a *dichotomic search*. We sample the average time per iteration at five regularly spaced halo-sizes, and recurse within the best interval.

Figure 5.20 shows the efficiency of the autotuning for eight applications and two input sizes each. Each bar represents the final speedup obtained across the given number of GPUs when accounting for the overhead of the online search phase and the outcome of each search strategy. Each application executes 100,000 iterations. The average speedup for applications running on two GPUs is 1.83x, and on four GPUs is 2.86x. Speedup increases with problem size, which indicates that communication is the limiting factor to scalability. 3D applications with an input size of 128^3 are not worth distributing across several devices, since the communication dominates over the gain of parallel computation.

For the online tuning, we observe that the overall performance is similar for the three searches. In most cases, *hill climbing* and the *dichotomic search* perform a little better than the *exhaustive search*. In particular, this is the case for larger problem sizes.

	Game Of Life		Swim		Himeno		Hyper	
	2048^2	4096^2	2048^2	4096^2	128^3	256^3	128^3	256^3
Offline Search Parameters								
Number of GPUs	2	4	4	4	1	4	1	2
Data Placement	B/D	C/D	C/D	C/D	n/a	C/D	n/a	B/D
Exhaustive Search Speedup								
Search Phase Perf. (%)	94.43	95.44	90.67	87.26	n/a	91.83	n/a	91.13
Post Search Perf. (%)	100.00	100.00	100.00	100.00	n/a	100.00	n/a	100.00
Iterations to converge	36,550	145,350	18,490	72,250	n/a	550	n/a	2,100
Hill Climbing Search Speedup								
Search Phase Perf. (%)	71.11	94.96	89.12	90.63	n/a	92.41	n/a	98.34
Post Search Perf. (%)	83.19	99.98	100.00	95.35	n/a	100.00	n/a	100.00
Iterations to converge	2,300	2,750	7,280	23,030	n/a	540	n/a	90
Dichotomic Search Speedup								
Search Phase Perf. (%)	97.90	95.53	91.24	87.93	n/a	94.02	n/a	91.67
Post Search Perf. (%)	99.36	96.78	93.69	95.35	n/a	100.00	n/a	100.00
Iterations to converge	4,260	9,400	5,510	8,430	n/a	380	n/a	540

Notes: *Data Placement Abbreviations* : (B) Blocking, (C) Circular, (D) Dual Cards, (S) Single Card
Search Phase Perf.: percentage of oracle performance obtained during the online tuning phase.
Post Search Perf.: percentage of oracle performance obtained after the online tuning phase.

Table 5.2: Performance of various search strategies.

In order to compare the searches more accurately, we break down the performance impact of the halo refinement phase and the outcome of the search in Table 5.2. We evaluate the quality and online performance overhead of our strategies with respect to the performance of an idealized *oracle* strategy. The oracle performance corresponds to the best performance found for the given application across the full optimization space, excluding tuning overheads. For each application, input size and search strategy, we report:

- the percentage of oracle performance obtained during the online tuning phase. This figure reflects the overhead introduced by our online scheme and the sub-optimal halos with which it experiments.
- the percentage of oracle performance obtained after the online tuning phase. This figure reflects the quality of the tuning outcome.
- the number of iterations to convergence. This captures the minimum number of iterations necessary for the search to converge. To increase the accuracy of the measurement and reduce the noise, each swapping is sampled ten times, making it expensive in terms of iterations for large halos.

The *exhaustive search* guarantees to find the optimal halo size eventually, which gives perfect post-tuning performance. It requires a very expensive search phase, but as the number of iteration increases, this search phase is amortized, giving this strategy a good overall performance. However, when the number of iterations is not high enough, the search does not have time to sample the entire space, and the overhead is much higher. This is the case for *Game Of Life* with a volume size of 4096, which we allowed to overrun beyond the standard 100,000 iterations in order to fully measure convergence. This effect is increased by an important performance difference across the full range of possible halo sizes, which increases the full search time, as shown in Section 5.5.3.

Hill climbing converges faster and is more efficient than the exhaustive search in most cases. For *Hyperthermia* with a volume size of 256, the convergence is achieved twenty times faster. However, for *Himeno*, it is almost as expensive as the exhaustive search because the optimal halo is large. Therefore, the high-quality result is not surprising in this case. Furthermore, it cannot cope well with bumpy search spaces. For example, with *Game of Life* and a volume of 2048, hill climbing stopped in a local minimum, missing the true optimal halo. Despite a fast convergence, the overall performance converges to the outcome of the search phase, which is in this case worse than performing a complete search.

Dichotomic search is the fastest to converge, but gives merely an approximation of the best halo size. As it recursively samples only a few points and makes a search using the extremities of each range, the result of the search could be quite far from the value of the optimal halo size. However, it does not stop on the first local minima like the hill climbing strategy. For applications with a sizable input data, this strategy is essential to navigate through the large parameter space in a small number of iterations. The number of iterations necessary to sample a halo size is relative to its size, as it affects the swapping frequency, which is why non iterative searches are more performant for large ranges. Furthermore, as demonstrated by the synthetic benchmarks, the optimal halo size tends to increase as the compute time decreases, making non linear searches like the *dichotomic search* well adapted to explore large parameter spaces with a small overhead at the cost of having only an approximation of the optimal halo.

For the 3D applications, *Himeno* and *Hyperthermia*, there is little difference between the three searches. This is due to the high number of iterations and the relatively small range of possible halo sizes. All three searches perform well and find a point close to the opti-

mal halo size in a small number of iterations. The overall speedup converges to this value, amortizing the search overhead. For larger problem sizes, the same differences as the 2D applications would arise with a greater magnitude, as the amount of redundant computation increases much faster when the dimensionality of the cut increases.

5.6 SUMMARY

The main contribution of this chapter is the discovery of the heterogeneous nature of seemingly homogeneous systems. Even when resources seem evenly distributed from a programmatic point of view, the hardware implementation can introduce imbalance, which can be adjusted by an auto-tuning mechanism to improve performance. This has been demonstrated with respect to the PCI bus connecting multiple GPUs.

This chapter also presented PARTANS, a framework for distributed stencil computation. By restricting the domain to a specific computational pattern, it is possible to look in detail at the main tuning parameters of a heterogeneous system and devise optimization strategies. This can be abstracted from a novice user by hiding the complexity and even the parallel nature of the underlying system.

The following chapter concludes this thesis by providing a summary of the contributions and a critical analysis, as well as future work.

6 CONCLUSION

This thesis has introduced two complementary techniques to tackle programmability challenges for heterogeneous systems. More specifically, the newly developed methods reduce the paradoxical disparity between productivity and performance in the OpenCL programming model. Chapter 4 provided a transparent solution for under-utilization of the model. A self-optimizing extension is added to OpenCL to replace costly manual optimizations and extend its applicability. The other extreme – abusing the model for performance – was addressed in Chapter 5. While not expressible within the model, hardware sensitive optimizations are uncovered and exploited within it.

The main contributions of these two approaches are summarized in Section 6.1. They are critically analyzed in Section 6.2. Section 6.3 discusses future developments of this work. Section 6.4 concludes by putting the contributions in the context of the overall problem they addressed.

6.1 CONTRIBUTIONS

This section summarizes the main contributions presented in the previous two chapters. Helium and Partans both provide new insights on heterogeneous system optimization and implement complementary schemes to address the programmability of such systems in terms of productivity, portability and performance. Their individual contributions are further detailed below.

6.1.1 *Transparent Dynamic Optimizations with Helium*

Chapter 4 presented and evaluated the *delay-optimize-replay* strategy implemented in Helium. Deployed as a transparent layer between any OpenCL application and the vendor implementation, an optimizer combining static and dynamic analysis techniques can per-

form aggressive compute-flow sensitive transformations, like merging computation across kernels or parallelizing independent tasks. This not only replaces an extremely tedious and error-prone manual optimization process but also extends the applicability of the OpenCL model by dramatically increasing performance of highly dynamic applications.

We demonstrated this technique on a set of benchmarks, comparing generic and unoptimized applications running with Helium to the same application manually optimized by an expert. We showed that this technique is lightweight enough to preserve performance on non-optimizable applications, sound enough to perform the same static optimizations as a human expert, and adaptive enough to take advantage of a dynamic context to surpass human experts by up to 80%.

6.1.2 *Multi-GPU Stencil Computation with Partans*

Most high-level and portable optimization techniques, including Helium, optimize against the model rather than the underlying hardware features. However, by restricting the domain to do a careful parameter space exploration, performance can be further enhanced in an equally portable way by taking advantage of the architecture. Partans was developed as a high productivity and high-performance framework for distributed stencil computation. A clean high-level interface allows the user to specify only the stencil specifications, and our automatic optimizer can fine-tune the optimization parameters.

Well known computation and communication tradeoffs can be balanced at runtime with low overhead for a multi-GPU system using auto-tuning techniques. The space was further explored using synthetic benchmarks, focusing purely on communication. The findings highlight that subtle architectural differences introduce heterogeneity in a seemingly homogeneous system. Across multiple benchmarks, the topology of the PCIe interconnecting four indistinguishable GPUs was shown to impair communication for certain configurations. Integrating this knowledge in an auto-tuner further improved performance by 25% compared to a non-aware optimizer.

Together, Helium and Partans provide complementary approaches to optimize an application against an abstract model to make good use of a parallel paradigm like OpenCL; and against hidden architectural artifacts introduced by increasingly complex systems.

They both contributed in distinct ways to improve performance, portability and productivity for heterogeneous systems.

6.2 CRITICAL ANALYSIS

While the contributions presented significantly improve the programmability of heterogeneous systems, some aspects and methods of the research need to be scrutinized and critically analyzed. This section lists and discusses these issues.

6.2.1 *Dynamic Optimizations of Data Flow in OpenCL*

OPTIMIZATION OVERHEADS The delay-optimize-replay mechanism used by HELIUM allows it to perform efficient runtime optimizations resulting in a clear performance gain. However, this technique might introduce overheads. First, executing commands lazily instead of eagerly might prevent the host and the device applications to perform computation in parallel. However, in practice, most of the time the host actively waits for the device after issuing a set of commands. Second, the complexity of the graph based optimizations applied on the graph might not scale to very large graphs. While HELIUM has been benchmarked on a few thousand delayed commands in the largest benchmark, scheduling tens of thousands of nodes might expose the overhead of the scheduler. This could be addressed by dividing the task graph in manageable schedules and improving the optimization heuristics.

DIFFERENT ARCHITECTURES The experiments presented in Chapter 4 only evaluated a large scale dedicated GPU architecture. While the optimization set presented in the framework are applicable to any architecture, their efficiency would certainly vary depending on the target. Eliminating global memory accesses is a particularly efficient optimization on GPUs, but cache effects would lessen the benefits on CPUs, where other optimizations such as vectorization might be more efficient. A careful comparative study would allow each optimization to be prioritized for each architecture and enable a more portable and adaptive optimizer.

HOST PROGRAM ANALYSIS Performing a static analysis of the host program as well as the device code would allow fine grained analysis of the memory accesses. While pointer and escape analysis are undecidable problems, a restricted analysis as described by Jablin et al. [Jab+11] is enough to determine the synchronization points and communication patterns in an OpenCL application. This could be used to simplify the task graphs, remove spurious synchronization and optimize communication between the host and the device. However, this would break the interoperability of HELIUM with pre-compiled binaries and change the optimization strategy from purely dynamic to a two-stage optimization.

LIMITS OF KERNEL FUSION The optimizer in HELIUM has primitive heuristics to determine whether kernels are fusable, and always merges them if they share neighboring data. While this approach generates good performance improvement in most cases, it might also dramatically degrade it for some kernels. for example, Increasing the amount of local memory required by aggregating local buffers might impact the scheduler on GPUs and result in a much lower number of concurrent threads. This can also be observed for kernels using a high amount of registers: the load elimination pass is forcibly applied but is known to cause extended live ranges and increase the overall number of registers. On some GPUs it might cause the same resource contention effects as the increased usage of local memory and decrease the number of concurrent workgroups. Predicting these effects in the optimizer to prevent overeager fusion would have to be investigated.

REUSABILITY OF SPECIALIZED CODE The JIT compiler in HELIUM maintains a code-cache to store specialized code in order to reuse it whenever possible, thereby lowering compilation overheads. However, the heuristics used for code staging are too simplistic: like kernel fusion, the optimizer always specializes the code if it might yield good performance. There is an important tradeoff which it not being considered: sacrificing a bit of performance for the sake of re-usability. Overly specialized code is less likely to be used again, and some transformations less intrusive than others. This is a common problem for many runtime staging platforms, and existing solutions could be adapted and integrated to HELIUM to improve its performance.

6.2.2 *Distributed Stencil Computations on Heterogeneous Systems*

SINGLE DEVICE OPTIMIZATIONS The evaluation of PARTANS primarily investigated communication optimization. The framework implements some optimizations to carry out the computation efficiently but does not implement most of the optimizations investigated by prior work and described in Section 3.3. The space was explored thoroughly using synthetic loads to account for shortened compute time. However, some conflicts might arise between optimizing the distribution across devices and localized optimizations, such as different volume transformations or data layout. The auto-tuner would have to take this into account and mitigate all parameters collectively.

HIGHER TIME-ORDER STENCILS The experimentation explored stencils of multiple space orders but did not investigate multiple time orders. Maintaining the representation for several timesteps would dramatically change the communication patterns and shift the computation to communication ratios. This would probably exacerbate the heterogeneous nature of the interconnect mediums and strengthen the claims. However, high time-order stencils are not well suited to GPUs since global memory is a very limited resource. The current investigation with a single order in time is already reaching this limit. A higher order would have considerably shrunk the research space.

MULTIDIMENSIONAL CUTS The slicing investigated in PARTANS is limited to a single dimension. This approach is unrealistic, particularly for large volumes with a low dimensionality. Existing distributed stencil applications generally adopt a multi-dimensional cut, which generates considerably more complex dependency graphs since a tile might depend on more than two neighbors. Implementing this would neither contradict nor reinforce the current findings: the data placement would still have an impact on performance and an auto-tuner would still need to investigate this. However, the search techniques would have to be adapted.

HETEROGENEOUS COMMUNICATION The parameter space explored by PARTANS proves that seemingly homogeneous systems like dedicated accelerators sharing the same PCI bus can be heterogeneous. However, it did not investigate heterogeneous interconnect in the

first place. Technology has already changed dramatically since this evaluation was performed. Accelerators can now share the same die as the CPU and use the same memory bus, newer motherboards support of mix of PCIe 2.0 and PCIe 3.0 slots, and vendors investigate high efficiency interconnect buses for peer-to-peer GPU communication like NVLink. These new technologies only increase the impact of data placement on performance. A careful investigation and benchmarking of the different communication channels would allow the implementation of a more complete machine learning algorithm deciding the optimal placement in a highly heterogeneous system.

6.2.3 *Combining Helium and Partans*

Paradoxically, Partans and Helium use the very same feature of the OpenCL model – the ability to manipulate tasks – to achieve exact opposite goals, yet both improve productivity and performance in complementary ways.

Helium showed that fusing tasks and eliminating all non-critical dependencies between them maximizes the parallelism inherent to the model and improve performance. Partans is doing the exact contrary: dividing tasks and adding non-critical dependencies to decrease the amount of parallelism in order to exploit hardware features.

Thus, in effect, applying Helium to Partans would counteract most of the optimization process and considerably degrade performance. More specifically, the range-aware horizontal fusion in Helium would undo the domain decomposition so carefully performed by Partans; and the dependency optimizations in Helium would quash Partans's efforts to regulate communication.

This dichotomy does not make the two approaches incompatible – just their current implementation. There is a delicate balance for the optimal amount of parallelism in a system. Not enough under-utilizes the resources. Too much inevitably creates congestions. Helium implements techniques to increase parallelism, Partans demonstrates how and when it should be harnessed. Hence these optimizations are complementary to balance heterogeneous system utilization. Finding this sweet spot, however, is a more difficult task akin to the phase ordering problem encountered by compilers.

6.3 FUTURE WORK

This section discusses how the two frameworks, PARTANS and HELIUM, could be extended in the future to provide additional features and explore new optimizations.

6.3.1 *Optimizing Tasking Model for OpenCL*

TASK FISSION In order to maximize the optimization potential of fusion, it would be necessary to first split existing kernels into indivisible units. This would allow our current optimization technique to reassemble only the parts of the kernel maximizing the benefits of kernel fusion. A similar strategy is already implemented in **StreamIt** [Gor+02], where they define *vertical fission* as splitting an operator and *horizontal fission* as splitting the domain.

INTEGRATING COMPILER TRANSFORMATIONS Since the *delay-optimize-replay* considerably decouples the host and device applications, a plethora of existing compiler transformations could be transparently applied with very low overheads. Some, like thread coarsening [MDO14], have shown to be very effective to optimize single kernels. Because HELIUM combines static and runtime analysis, much more is known about the execution context of each kernel instance, which can be used to drive aggressive optimizations like introducing local memory caches.

DYNAMIC RE-TARGETING For large task graphs with independent sub-graphs, it is be possible to split a large graph and dispatch the computation to multiple devices. The additional communication operations between the devices could be inferred automatically from the dataflow graph analyzed by HELIUM. Scheduling heuristics like the one described by Grewe, Wang, and O'Boyle [GWO13] can be used to determine which device is best suited for any given task; or to distribute the computation across multiple devices.

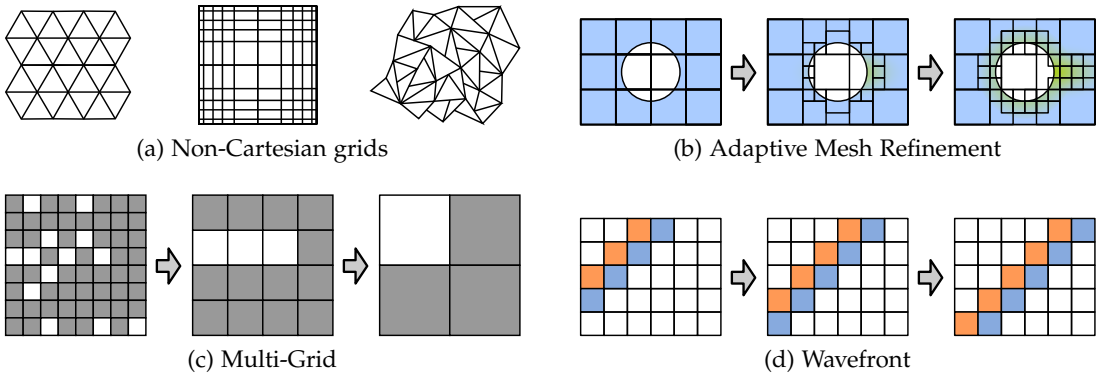


Figure 6.1: Problems derived from the stencil computation pattern. The domain can be non-Cartesian(a). The grid can also dynamically adapt to computation(b) or reduce at each iteration(c). Other patterns use only a subset of the domain, like the wavefront pattern(d).

6.3.2 *Distributed Stencil Computation*

MULTI-LEVEL PARALLELISM Stencil computations have been parallelized on various systems: existing research explored multi-node clusters using MPI and multi-processor devices such as CPUs and GPUs; and we explored multi-device distribution. Combining all these approaches at once for a cluster of nodes composed of multiple parallel devices would prove very challenging since the layers cannot be optimized and tuned in isolation but together since they are not independent.

COMPUTE DOMAIN EXTENSIONS There are many other forms of stencil computations which are not explored in Chapter 5. Figure 6.1 shows other types of problems also using a nearest computation pattern. Figure 6.1a shows examples of non-Cartesian domains: they can be expressed using triangular, rectilinear or curvilinear volumes, or even unstructured grids. Grids can also be non-uniform, which is used for optimization of sparse problems where only a small subset of the domain yields interesting computation. This process, shown in Figure 6.1b, is called *adaptive mesh refinement*.

DERIVED SKELETONS The output and input domains may be of different size, and iteratively repeating the computation lowers the resolution and solves a more coarse problem; these are known as *multi-grid methods*(Figure 6.1c). Finally, the stencil operator can be mapped to a subset of the domain in a structured way, like the *wavefront* pattern (Figure 6.1d). While stencil optimizations can be applied in this case, other tuning parameters specific to this problem must be taken into consideration and integrated to the auto-tuner.

6.4 FINAL REMARKS

Heterogeneous systems are here to stay, and will only grow in diversity and complexity. The emerging abstraction models which set out to tackle the resulting programmability challenges show contrasted results. On the one hand, the accessible portability they provide by-design lifts a considerable burden. On the other hand, their genericity cannot quite profit from all intricacies of the hardware. Programmers either take it upon themselves to bridge this gap, or overly rely on the models to do it for them. Either way, they resort to all sorts of under/mis/ab-uses of portable programming models – causing an abyssal chasm between productivity and performance.

The techniques presented in this thesis demonstrate that this need not be – or at least not anywhere near the current extent. Under-utilization of the heterogeneous programming model is addressed with a safe, transparent self-optimizing layer. At the other end of the spectrum, misuses are replaced with a portable auto-tuner. In both cases, these new methods considerably enhance performance and productivity, and preserve portability.

Unfortunately, they cannot reach the best possible performance on most systems – and neither would any other optimization technique in the same context. The reason is simply because this point is not within reach for the current programming models. Programmers knowing this routinely violate model requirements to squeeze a bit more performance – abandoning productivity and portability altogether.

In conclusion, programmability of heterogeneous systems is still a challenge. However, we are closer than ever before to reconciling productivity, portability and performance. Time will tell if the key to finally tame heterogeneity lies in better programming models or better ways to use of them – or both.

A EXAMPLE OF OPENCL APPLICATION

Canny Edge Detection in OpenCL

```
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>
#include <vector>

// Device source code
const std::string code = R"CODE(
#define IDX(X,Y,WIDTH) ((X) + ((Y) * (WIDTH)))
#define off(y,x) (in[IDX(tid0+(x), tid1+(y), n0)])
#define PREAMBLE \
  const int tid0 = get_global_id(0), tid1 = get_global_id(1);\
  const int n0 = get_global_size(0), n1 = get_global_size(1);\
  const int idx = IDX(tid0, tid1, n0)

kernel void gaussian(const global float * in, global float * out) {
  PREAMBLE;
  out[idx] = (tid0 > 1 && tid0 < n0-2 && tid1 > 1 && tid1 < n1-2) ?
    (  off(-1,-1) + 2*off(-1,+0) +   off(-1,+1) +
       2*off(+0,-1) + 4*off(+0,+0) + 2*off(+0,+1) +
       off(+1,-1) + 2*off(+1,+0) +   off(+1,+1) ) / 16.0f : 0.0f;
}

kernel void gradient_x(const global float * in, global float * out) {
  PREAMBLE;
  out[idx] = (tid0 > 0 && tid0 < n0-1 && tid1 > 0 && tid1 < n1-1) ?
    off(-1,1)-off(-1,-1)+2*(off(0,1)-off(0,-1))+off(1,1)-off(1,-1) : 0.0f;
}

kernel void gradient_y(const global float * in, global float * out) {
  PREAMBLE;
  out[idx] = (tid0 > 0 && tid0 < n0-1 && tid1 > 0 && tid1 < n1-1) ?
    off(-1,-1)-off(1,-1)+2*(off(-1,0)-off(1,0))+off(-1,1)-off(1,1) : 0.0f;
}

float retabs(const float f) { return f < 0 ? -f : f; }

kernel void magnitude(const global float * in1,
                      const global float * in2,
                            global float * out) {
  PREAMBLE; out[idx] = retabs(in1[idx]) + retabs(in2[idx]);
}
)CODE";

```

```
int main(){
  try {
    using namespace cl;
    cl_int err = CL_SUCCESS;

    // parameters
    constexpr ::size_t width = 8000, height = 6000;
    constexpr ::size_t pixels = width * height;
    constexpr ::size_t buf_size = pixels * sizeof(float);

    NDRange none{NullRange}, range{width, height}; // grid size and offsets
    std::vector<float> img(pixels), res(pixels); // host side data

    // Initialize the device
    std::vector<Platform> plats;
    Platform::get(&plats);
    if (plats.size() == 0) return -1;

    cl_context_properties properties[] =
      { CL_CONTEXT_PLATFORM, cl_context_properties(plats[0]()), 0};
    Context ctx{CL_DEVICE_TYPE_ALL, properties};
    auto devices = ctx.getInfo<CL_CONTEXT_DEVICES>();

    CommandQueue queue(ctx, devices[0], 0, &err); // Create a command queue

    Program program{ctx,{1,std::make_pair(code.data(),code.size())}};
    program.build(devices); // Create and build the program

    Kernel // Create Kernel Objects
      k_gaussian{program, "gaussian", &err},
      k_gradient_x{program, "gradient_x", &err},
      k_gradient_y{program, "gradient_y", &err},
      k_magnitude{program, "magnitude", &err};
    KernelFunctor // Create Kernel functors
      f_gaussian{k_gaussian, queue, none, range, none},
      f_gradient_x{k_gradient_x, queue, none, range, none},
      f_gradient_y{k_gradient_y, queue, none, range, none},
      f_magnitude{k_magnitude, queue, none, range, none};
    Buffer // allocate the device memory
      b_img{ctx, CL_MEM_READ_WRITE, buf_size, nullptr, &err},
      b_gaussian{ctx, CL_MEM_READ_WRITE, buf_size, nullptr, &err},
      b_gradx{ctx, CL_MEM_READ_WRITE, buf_size, nullptr, &err},
      b_grady{ctx, CL_MEM_READ_WRITE, buf_size, nullptr, &err},
      b_mag{ctx, CL_MEM_READ_WRITE, buf_size, nullptr, &err};

    // enqueue the kernels
    f_gaussian(b_img, b_gaussian);
    f_gradient_x(b_gaussian, b_gradx);
    f_gradient_y(b_gaussian, b_grady);
    f_magnitude(b_gradx, b_grady, b_mag);

    // read the result
    queue.enqueueReadBuffer(b_mag, CL_TRUE, 0, buf_size, &res[0]);
  } catch (cl::Error err) { return err.err(); }
}
```

BIBLIOGRAPHY

[AMP14] Fernando Alexandre, Ricardo Marques, and Hervé Paulino. "On the Support of Task-parallel Algorithmic Skeletons for multi-GPU Computing." In: *Proc. of the 29th Annual ACM Symp. on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: ACM, 2014 (Cited on pages 46 and 62).

[Arn+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. "Adaptive Optimization in the JalapeNO JVM." In: *Proc. of the 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '00. Minneapolis, Minnesota, USA: ACM, 2000 (Cited on page 49).

[AT+98] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. "Fast, Effective Code Generation in a Just-in-time Java Compiler." In: *Proc. of the 19th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998 (Cited on page 49).

[Aug+10] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. "Data-Aware Task Scheduling on Multi-accelerator Based Platforms." In: *Proc. of 16th Int. Conf. on Parallel and Distributed Systems*. ICPADS. Dec. 2010 (Cited on page 46).

[Bec+04] Olav Beckmann, Alastair Houghton, Michael Mellor, and PaulH.J. Kelly. "Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation." English. In: *Domain-Specific Program Generation*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Vol. 3016. Lecture Notes in Computer Science. 2004 (Cited on page 50).

[BP10] Tobias Brandvik and Graham Pullan. "SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms." In: *Proc. of the 2010 10th IEEE Int. Conf. on Computer and Information Technology*. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010 (Cited on pages 57, 58, and 59).

[BPB12] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. "Tiling Stencil Computations to Maximize Parallelism." In: *Proc. of the 2012 ACM/IEEE Conf. on Supercomputing*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012 (Cited on page 55).

[Buc+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. "Brook for GPUs: Stream Computing on Graphics Hardware." In: *Proc. of the 31th Annual Conf. on Computer Graphics and Interactive Techniques*. SIGGRAPH '04. Los Angeles, California: ACM, 2004 (Cited on pages 10, 26, and 41).

[Col04] Murray Cole. "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming." In: *Parallel Comput.* 30.3 (Mar. 2004) (Cited on page 24).

[CSB11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "Automatic code generation and tuning for stencil kernels on modern shared memory architectures." In: *Computer Science - Research and Development* (2011) (Cited on pages 59 and 126).

[Cud] *CUDA Specifications*. NVIDIA Corporation. 2014. URL: `http://docs.nvidia.com/cuda/` (Cited on page 26).

[Dar99] Alain Darte. "On the Complexity of Loop Fusion." In: *Proc. of the 1999 Int. Conf. on Parallel Architectures and Compilation Techniques*. PACT '99. Washington, DC, USA: IEEE Computer Society, 1999 (Cited on page 53).

[Dat+08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures." In: *Proc. of the 2012 ACM/IEEE Conf. on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008 (Cited on page 56).

[DEK11] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. "Auto-tuning SkePU: A Multi-backend Skeleton Programming Framework for multi-GPU Systems." In: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. IWMSE '11. Honolulu, HI, USA: ACM, 2011 (Cited on page 46).

[Den] *Project Denver processor to usher in a new era of computing*. NVIDIA Corporation. 2011. URL: `http://blogs.nvidia.com/blog/tag/project-denver/` (Cited on page 11).

[Den+74] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. "Design of ion-implanted MOSFETs with very small physical dimensions." In: *IEEE J. Solid-State Circuits* (1974) (Cited on page 1).

[Dub+12] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. "Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers)." In: *Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012 (Cited on page 49).

[EG01] Robert A. van Engelen and Kyle A. Gallivan. *An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications*. 2001 (Cited on page 78).

[EK10] Johan Enmyren and Christoph W. Kessler. "SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems." In: *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*. HLPP '10. Baltimore, Maryland, USA: ACM, 2010 (Cited on page 42).

[EK12] Steffen Ernsting and Herbert Kuchen. "Algorithmic Skeletons for Multi-core, multi-GPU Systems and Clusters." In: *Int. J. High Perform. Comput. Netw.* 7.2 (Apr. 2012) (Cited on page 42).

[FHM99] Antoine Fraboulet, Guillaume Huard, and Anne Mignotte. "Loop Alignment for Memory Accesses Optimization." In: *Proceedings of the 12th International Symposium on System Synthesis*. ISSS '99. Washington, DC, USA: IEEE Computer Society, 1999 (Cited on page 94).

[FS05] Matteo Frigo and Volker Strumpen. "Cache Oblivious Stencil Computations." In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts: ACM, 2005 (Cited on page 53).

[Fuh+14] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas Schulthess. "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models." In: *Supercomputing frontiers and innovations* 1.1 (2014) (Cited on pages 57 and 60).

[GAK03] Georgios Goumas, Maria Athanasaki, and Nectarios Koziris. "An Efficient Code Generation Technique for Tiled Iteration Spaces." In: *IEEE Transactions on Parallel and Distributed Systems* 14 (2003) (Cited on page 53).

[Gor+02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. "A Stream Compiler for Communication-exposed Architectures." In: *Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. San Jose, California, 2002 (Cited on page 149).

[Gra] *Graal project*. `http://openjdk.java.net/projects/graal`. OpenJDK. 2013 (Cited on page 49).

[Gro+13] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. "Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles." In: *Proc. of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. GPGPU-6. Houston, Texas: ACM, 2013 (Cited on page 55).

[Gro+14] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. "Hybrid Hexagonal/Classical Tiling for GPUs." In: *Proc. of Annual IEEE/ACM Int. Symp. on Code Generation and Optimization*. CGO '14. Orlando, FL, USA: ACM, 2014 (Cited on page 55).

[Gro08] Khronos Group. *Khronos Launches Heterogeneous Computing Initiative*. June 2008. URL: `http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/` (Cited on page 27).

[GWO13] D. Grewe, Zheng Wang, and M.F.P. O'Boyle. "Portable mapping of data parallel programs to OpenCL for heterogeneous systems." In: *Code Generation and Optimization (CGO), 2013 IEEE/ACM Int. Symp. on*. 2013 (Cited on pages 48 and 149).

[Göd+08] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Hilmar Wobker, Christian Becker, and Stefan Turek. "Using GPUs to Improve

Multigrid Solver Performance on a Cluster." In: *Int. Journal of Computational Science and Engineering (IJCSE)* 4.1 (2008) (Cited on page 56).

[Han+11] Dongni Han, Shixiong Xu, Li Chen, and Lei Huang. "PADS: A Pattern-Driven Stencil Compiler-Based Tool for Reuse of Optimizations on GPGPUs." In: *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. 2011 (Cited on pages 57 and 60).

[Har15] Mark Harris. *CUDA 7 Release Candidate Feature Overview*. 2015. URL: http://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/ (Cited on page 49).

[HB10] Jared Hoberock and Nathan Bell. *Thrust: A Parallel Template Library*. 2010. URL: http://thrust.github.io (Cited on page 27).

[HPS12] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. "High-performance Code Generation for Stencil Computations on GPU Architectures." In: *Proc. of the 26th ACM Int. Conf. on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: ACM, 2012 (Cited on pages 57, 58, and 59).

[HR10] A. Hilton and A. Roth. "BOLT: Energy-efficient Out-of-Order Latency-Tolerant execution." In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th Int. Symp. on*. 2010 (Cited on page 27).

[Int14] Intel. *Intel Xeon Processor E7 v2 Family Product Brief*. 2014. URL: www.intel.com/xeonE7 (Cited on page 10).

[IT88] F. Irigoin and R. Triolet. "Supernode Partitioning." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988 (Cited on page 53).

[Jab+11] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. "Automatic CPU-GPU Communication Management and Optimization." In: *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA, 2011 (Cited on pages 47 and 146).

[Jan+11] Byunghyun Jang, D. Schaa, P. Mistry, and D. Kaeli. "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures." In: *Parallel and Distributed Systems, IEEE Transactions on* 22.1 (2011) (Cited on page 44).

[Ji+12] Feng Ji, A.M. Aji, J. Dinan, D. Buntinas, P. Balaji, Wu chun Feng, and Xiaosong Ma. "Efficient Intranode Communication in GPU-Accelerated Systems." In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*. 2012 (Cited on page 62).

[Kam+05] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. "Impact of modern memory subsystems on cache optimizations for stencil computations." In: *Proc. of the 2005 workshop on Memory system performance*. MSP '05. Chicago, Illinois: ACM, 2005 (Cited on page 60).

[Kam+10] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. "An auto-tuning framework for parallel multicore stencil computations." In: *Parallel Distributed Processing (IPDPS), 2010 IEEE Int. Symp. on*. 2010 (Cited on page 60).

[KG97] Andreas Krall and Reinhard Grafl. "CACAO - A 64-bit JavaVM just-in-time compiler." In: *Concurrency: Practice and Experience* 9.11 (1997). URL: http://dx.doi.org/10.1002/(SICI)1096-9128(199711)9:11<1017::AID-CPE347>3.0.CO;2-0 (Cited on page 49).

[Kim+11] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs." In: *Proc. of the 16th ACM Symp. on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011 (Cited on page 43).

[KM94] Ken Kennedy and Kathryn S. McKinley. "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution." In: *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. London, UK, UK: Springer-Verlag, 1994 (Cited on page 53).

[KSG13] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. "dOpenCL: Towards Uniform Programming of Distributed Heterogeneous Multi-/Many-core Systems." In: *J. Parallel Distrib. Comput.* 73.12 (Dec. 2013) (Cited on page 43).

[Lee+10] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. "An OpenCL Framework for Heterogeneous Multicores with Local Memory." In: *Proc. of the 19th Int. Conf. on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, 2010 (Cited on page 43).

[Lee+11] Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. "An OpenCL Framework for Homogeneous Manycores with No Hardware Cache Coherence." In: *Parallel Architectures and Compilation Techniques (PACT), 2011 Int. Conf. on*. 2011 (Cited on page 43).

[LFC13] Thibaut Lutz, Christian Fensch, and Murray Cole. "PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems." In: *ACM Transactions on Architecture and Code Optimization*. TACO 9.4 (Jan. 2013) (Cited on page 6).

[LFC15] Thibaut Lutz, Christian Fensch, and Murray Cole. "Helium: A Transparent Inter-Kernel Optimizer for OpenCL." In: *Proc. of Workshop on General Purpose Processing Using GPUs*. GPGPU-8. San Francisco, CA, USA: ACM, 2015 (Cited on page 6).

[LG14] Thibaut Lutz and Vinod Grover. "LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms." In: *Proc. of the 8th Workshop on General Purpose Processor Using Graphics Processing Units*. FHPC '14. Gothenburg, Sweden: ACM, 2014 (Cited on page 6).

[LME09] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization." In: *Proc. of*

the 14th ACM Symp. on Principles and Practice of Parallel Programming. PPoPP '09. Raleigh, NC, USA: ACM, 2009 (Cited on page 48).

[LZS09] Yixun Liu, E.Z. Zhang, and Xipeng Shen. "A cross-input adaptive framework for GPU program optimizations." In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009 (Cited on page 45).

[Mar+11] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers." In: *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. 2011 (Cited on page 57).

[Mar+13] Ricardo Marques, HervÃ© Paulino, Fernando Alexandre, and PedroD. Medeiros. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations." English. In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Vol. 8097. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_86 (Cited on page 46).

[MC11] Helmar Burkhart Matthias Christen Olaf Schenk. "Patus: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures." In: *IPDPS*. 2011 (Cited on pages 57, 58, 59, and 127).

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. "Improving Data Locality with Loop Transformations." In: *ACM Trans. Program. Lang. Syst.* 18.4 (July 1996) (Cited on page 53).

[MDO13] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. "A Large-scale Cross-architecture Evaluation of Thread-coarsening." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013 (Cited on page 45).

[MDO14] Alberto Magni, Christophe Dubach, and Michael O'Boyle. "Automatic Optimization of Thread-coarsening for Graphics Processors." In: *Proc. of the 23rd Int. Conf. on Parallel Architectures and Compilation*. PACT. Edmonton, AB, Canada, 2014 (Cited on pages 45 and 149).

[Mic09] Paulius Micikevicius. "3D finite difference computation on GPUs using CUDA." In: *Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-2. Washington, D.C.: ACM, 2009 (Cited on page 55).

[Mis+11] Perhaad Mistry, Chris Gregg, Norman Rubin, David Kaeli, and Kim Hazelwood. "Analyzing Program Flow Within a Many-kernel OpenCL Application." In: *Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-4. Newport Beach, California, 2011 (Cited on page 47).

[MKR07] Tom Mertens, Jan Kautz, and Frank Van Reeth. "Exposure Fusion." In: *Proc. of Pacific Conf. on Computer Graphics and Applications*. 2007 (Cited on page 103).

[Moo65] Gordon E. Moore. "Cramming more components onto integrated circuits." In: *Electronics* 38.8 (1965) (Cited on page 1).

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. pp. 89-90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012 (Cited on page 36).

[MS11] Jiayuan Meng and Kevin Skadron. "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations." In: *Int. Journal of Parallel Programming* 39 (1 2011) (Cited on page 55).

[Ngu+10] A. Nguyen, N. Satish, J. Chhugani, Changkyu Kim, and P. Dubey. "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs." In: *High Performance Computing, Networking, Storage and Analysis (SC), 2010 Int. Conf. for*. 2010 (Cited on page 55).

[Opea] *The OpenACC Application Programming Interface, Version 1.0*. OpenACC Working Group. 2011. URL: http://www.openacc-standard.org/ (Cited on page 27).

[Opeb] *The OpenCL Specification version 1.2*. 19th ed. Khronos OpenCL Working Group. Nov. 2012. URL: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (Cited on page 27).

[PF10] Everett H. Phillips and Massimiliano Fatica. "Implementing the Himeno Benchmark with CUDA on GPU clusters." In: *Proc. of the 24th IEEE Int. Parallel and Distributed Processing Symp.* Apr. 2010 (Cited on page 127).

[PG14] Prasanna Pandit and R. Govindarajan. "Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices." In: *Proc. of Annual IEEE/ACM Int. Symp. on Code Generation and Optimization*. CGO '14. Orlando, FL, USA: ACM, 2014 (Cited on page 43).

[Phi] *Intel Xeon Phi Coprocessor*. Intel Corporation. URL: http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner (Cited on pages 2 and 11).

[Pol] *Intel Research Advances 'Era Of Tera'*. Intel Corporation. URL: http://www.intel.com/pressroom/archive/releases/2007/20070204comp.htm (Cited on page 2).

[Pur+02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. "Ray Tracing on Programmable Graphics Hardware." In: *Proc. of the 29th Annual Conf. on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, Texas: ACM, 2002 (Cited on page 10).

[Rei07] James Reinders. *Intel Threading Building Blocks*. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007 (Cited on page 25).

[RHG15] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. "Forma: A DSL for Image Processing Applications to Target GPUs and Multi-core CPUs." In: *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*. GPGPU 2015. San Francisco, CA, USA: ACM, 2015 (Cited on pages 57 and 58).

[RIF01] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. "Cactus Application: Performance Predictions in Grid Environments." In: *In Proc. of European Conf. on Parallel Computing (EuroPar) 2001*. 2001 (Cited on pages 57 and 60).

[RK+12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines." In: *ACM Trans. Graph.* 31.4 (July 2012) (Cited on page 59).

[RK+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines." In: *Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013 (Cited on page 59).

[Rom+13] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. "Surgical Precision JIT Compilers." In: *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2013 (Cited on page 49).

[Seo+13] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. "Automatic OpenCL Work-group Size Selection for Multicore CPUs." In: *Proc. of the 22nd Int. Conf. on Parallel Architectures and Compilation Techniques*. PACT '13. Edinburgh, Scotland, UK: IEEE Press, 2013 (Cited on page 46).

[SF12] Kazuki Sakamoto and Tomohiko Furumoto. "Grand Central Dispatch." English. In: *Pro Multithreading and Memory Management for iOS and OS X*. Apress, 2012 (Cited on page 25).

[SKG11] M. Steuwer, P. Kegel, and S. Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming." In: *IEEE Int. Symp. on Parallel and Distributed Processing Workshops and Phd Forum*. IPDPSW. 2011 (Cited on page 42).

[SL+07] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. "Sketching Stencils." In: *Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007 (Cited on page 60).

[SL05] John Paul Shen and Mikko H. Lipasti. *Modern processor design : fundamentals of superscalar processors*. Index. Boston: McGraw-Hill Higher Education, 2005 (Cited on page 24).

[SLH12] I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. "DL: A data layout transformation system for heterogeneous computing." In: *Innovative Parallel Computing*. inPar. 2012 (Cited on pages 21 and 44).

[Spi] *The SPIR Specification version 1.2*. 1st ed. Khronos SPIR Working Group. Jan. 2014. URL: `http://www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf` (Cited on page 35).

[SSH10] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. "Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications." In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, 2010 (Cited on page 44).

[Ste+14] Michel Steuwer, Michael Haidl, Stefan Breuer, and Sergei Gorlatch. "High-Level Programming of Stencil Computations on Multi-GPU Systems Using the SkelCL Library." In: *Parallel Processing Letters* 24.03 (2014) (Cited on page 60).

[Tan+11] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. "The Pochoir Stencil Compiler." In: *Proc. of the Twenty-third Annual ACM Symp. on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011 (Cited on pages 57, 58, and 59).

[Top14] Top500. *TOP500 supercomputing sites*. 2014. URL: `http://www.top500.org` (Cited on page 11).

[USQ12] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. "Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality." In: *Proceedings of the 21st International Conference on Compiler Construction*. CC'12. Tallinn, Estonia: Springer-Verlag, 2012 (Cited on page 45).

[VS+09] Brian Van Straalen, John Shalf, Terry Ligocki, Noel Keen, and Woo-Sun Yang. "Scalability challenges for massively parallel AMR applications." In: *Proc. of the 2009 IEEE Int. Symp. on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009 (Cited on page 60).

[WL91] Michael E. Wolf and Monica S. Lam. "A Data Locality Optimizing Algorithm." In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, 1991 (Cited on page 53).

[Wono0] D. Wonnacott. "Using time skewing to eliminate idle time due to memory bandwidth and network limitations." In: *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. 2000 (Cited on page 54).

[Yan+12] Yang Yang, Hui-Min Cui, Xiao-Bing Feng, and Jing-Ling Xue. "A Hybrid Circular Queue Method for Iterative Stencil Computations on GPUs." English. In: *Journal of Computer Science and Technology* 27.1 (2012). URL: `http://dx.doi.org/10.1007/s11390-012-1206-3` (Cited on page 54).

[Zim93] Eugene V. Zima. "Recurrent Relations and Speed-up of Computations Using Computer Algebra Systems." In: *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*. DISCO '92. London, UK, UK: Springer-Verlag, 1993 (Cited on page 79).

[Zim95] Eugene V. Zima. "Simplification and Optimization Transformations of Chains of Recurrences." In: *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*. ISSAC '95. Montreal, Quebec, Canada: ACM, 1995 (Cited on page 79).

[ZM12] Yongpeng Zhang and Frank Mueller. "Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters." In: *Proc. of the 10th Int. Symp. on Code Generation and Optimization*. CGO '12. San Jose, California: ACM, 2012 (Cited on pages 56, 57, and 62).

[NVI99] NVIDIA Corporation. *GeForce 256: The World's First GPU*. 1999. URL: http://www.nvidia.co.uk/page/geforce256.html (Cited on page 10).